AFRL-IF-WP-TR-2003-1545

UNIFIED DEBUG ENVIRONMENT FOR ADAPTIVE COMPUTING SYSTEMS

Dr. Brad Hutchings
Dr. Brent Nelson
Dr. Mike Wirthlin
Dr. Doran Wilde

Brigham Young University
Department of Electrical and Computer Engineering
459 Clyde Building
Provo, UT 84602

SEPTEMBER 2003

Final Report for 24 June 1999 – 31 July 2003

Approved for public release; distribution is unlimited.

STINFO FINAL REPORT

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

NOTICE

USING GOVERNMENT DRAWINGS, SPECIFICATIONS, OR OTHER DATA INCLUDED IN THIS DOCUMENT FOR ANY PURPOSE OTHER THAN GOVERNMENT PROCUREMENT DOES NOT IN ANY WAY OBLIGATE THE US GOVERNMENT. THE FACT THAT THE GOVERNMENT FORMULATED OR SUPPLIED THE DRAWINGS, SPECIFICATIONS, OR OTHER DATA DOES NOT LICENSE THE HOLDER OR ANY OTHER PERSON OR CORPORATION; OR CONVEY ANY RIGHTS OR PERMISSION TO MANUFACTURE, USE, OR SELL ANY PATENTED INVENTION THAT MAY RELATE TO THEM.

THIS REPORT IS RELEASABLE TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS). AT NTIS, IT WILL BE AVAILABLE TO THE GENERAL PUBLIC, INCLUDING FOREIGN NATIONS.

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.

/s/

AL SCARPELLI
Project Engineer/Team Leader
Embedded Info Sys Engineering Branch
Information Technology Division

/s/

JAMES S. WILLIAMSON, Chief
Embedded Info Sys Engineering Branch
Information Technology Division
Information Directorate

Do not return copies of this report unless contractual obligations or notice on a specific document requires its return.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
September 2003	Final	06/24/1999 – 07/31/2003

4. TITLE AND SUBTITLE
UNIFIED DEBUG ENVIRONMENT FOR ADAPTIVE COMPUTING SYSTEMS

5a. CONTRACT NUMBER
F33615-99-C-1502

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62301E

6. AUTHOR(S)
Dr. Brad Hutchings
Dr. Brent Nelson
Dr. Mike Wirthlin
Dr. Doran Wilde

5d. PROJECT NUMBER
ARPI

5e. TASK NUMBER
FT

5f. WORK UNIT NUMBER
0C

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Brigham Young University
Department of Electrical and Computer Engineering
459 Clyde Building
Provo, UT 84602

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Information Directorate
Air Force Research Laboratory
Air Force Materiel Command
Wright-Patterson Air Force Base, OH 45433-7334

DARPA / TTO
3701 North Fairfax Drive
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/IFTA

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-IF-WP-TR-2003-1545

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
Report contains color.

14. ABSTRACT ***(Maximum 200 Words)***

Adaptive computing systems (ACS) are hardware systems based around FPGA technology. Historically, design and debug tools for such systems have been based on ASIC technology. However, FPGA technology provides features which suggest different approaches be used for debug. For example, readback is a feature available in many FPGA devices which provides the ability to query an executing FPGA device for its entire internal state, providing unprecedented visibility into the executing design. The central element of the debug environment developed in this work is a debug services module (DSM). It provides a unified simulation/hardware execution debug environment which allows the user to debug the executing hardware in the context of the original design environment. This means signal values in the executing hardware can be viewed and manipulated using their original signal names from the design source. In addition, the DSM provides the following support for designs running on ACS platforms: checkpointing, multitasking, remote access, and interfacing with external high-level design tools. Finally, the DSM provides support for the automatic synthesis of debug circuitry to enable rapid instrumentation of FPGA designs for debug purposes. This report summarizes the debug system and a number of the experiments completed using it.

15. SUBJECT TERMS
Adaptive computing systems, FPGA, debug

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Alfred Scarpelli
Unclassified	Unclassified	Unclassified	SAR	82	19b. TELEPHONE NUMBER *(Include Area Code)* (937) 255-6548 x3603

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

Table of Contents

Section			Page
List of Figures			**vi**
List of Tables			**viii**
1	**Introduction and Motivation**		**1**
	1.1	Research Motivation	1
		1.1.1 State of the Art: 1999	1
		1.1.2 What Was Possible with 1999 Technology	2
	1.2	Overview of Proposed Research	2
2	**Report Overview**		**4**
3	**The Debug Services Manager (DSM)**		**5**
	3.1	The DSM	5
	3.2	The Unified Browser	6
	3.3	Unique Capabilities of the Debug Environment	7
	3.4	Internal Circuit Representation	9
	3.5	The Simulator	11
	3.6	Symbol Mapping Facility	16
	3.7	Checkpointing	18
		3.7.1 Checkpointing Architecture	18
		3.7.2 Checkpointing API	20
		3.7.3 Checkpointable and ExternallyUpdatable Interface	21
	3.8	Multitasking	21
		3.8.1 Hardware Context Switching	21
		3.8.2 Writeback	23
		3.8.3 Multitasking on SLAAC1-V	24
	3.9	Remote Access	26
	3.10	Platform Control	26
	3.11	Summary	26

4 The Platform Control Interface **27**

4.1 The Platform Independent Interface (PII) 27

4.2 Implementation of the Platform Independent Interface (PII) 28

4.3 The Hardware Control Interface (HCI) 30

4.3.1 Remote Access 30

4.4 The Platform Specific Interface (PSI) 33

4.5 Platform Specific GUI and CLI 34

4.6 Simulation Model 34

4.7 Native Code Library Interface 34

5 Debug Circuitry Synthesis **35**

5.1 A Bitstream-Modifiable Logic Analyzer 36

5.2 The Demonstration System 37

6 The Data Interchange Specification, Tools, and API **40**

6.1 Importing EDIF Designs into the Debug System Using the EdifParser 40

6.2 EDIF Coverage and Compatibility Issues 40

6.3 On-Line Documentation on Data Interchange 42

7 Using High-Level Tools with the DSM **43**

7.1 Source Level Debugger 43

7.2 A Debug Environment for Hybrid FPGA/CPU Systems 46

7.2.1 CPU-Specific Browser Windows 47

7.2.2 Extensions to the Hybrid Debug Environment 48

7.3 Integration of DSM to Ptolemy Modeling Environment 49

8 Use of the Unified Debug Environment on Real Applications **51**

8.1 Focus of Attention (FOA) 51

8.2 Contamination Distribution Indexer (CDI) 51

8.3 Custom Graphical Application Representation 52

9 On-line Documentation of the Unified Debug Environment **54**

10 Context Switching Devices Results Using the Sanders CSRC **57**

10.1 Investigate Context-Switched Approaches for Debug 57

10.2 Develop DSM support for CSRC 57

10.3 Test the CSRC Debug Support On CSRC-mapped Applications 58

10.4 CSRC Debug Demonstration 58

11 Conclusions and Future Work **63**

11.1 Future Work 63

12 References **65**

Appendices

A The DSM API's **67**

A.1 Debug Service Manager 67

A.1.1 The Simulator 67

A.1.2 The Hardware System (HWSystem) 67

A.1.3 Wire Value Interface 70

A.1.4 Circuit Structure 70

A.2 Platform Control Interface 71

A.2.1 Hardware Interface 71

A.2.2 Hardware Controller Interface 71

A.2.3 Readback Manager 72

List of Figures

Figure		Page
1	Organization of Debug Environment	5
2	Unified Browswer Screen Shot	7
3	Full-Adder Example Circuit	9
4	Interfaces Used In Conjunction With Simulation System	12
5	Simulation System Operating in Hardware Mode	13
6	Logical to Physical Mapping	16
7	Process of Logical-to-Physical Mapping	17
8	Example Use of Checkpointing	19
9	Checkpointing the DSM	19
10	Multitasking Three Applications	22
11	Four Steps of a Hardware Context Switch	23
12	State Restoration on User Flip-Flops	24
13	Client-Server Multitasking Architecture	25
14	API Layers Used in JHDL For Platform Control	28
15	Structure of the HardwareControlInterface	30
16	API Layers Used in JHDL For Remote Platform Control	33
17	Bitstream-Modifiable Logic Analyzer Core	36
18	Bitstream Configurable Trigger Logic	37
19	Instrumenting a User Circuit With a Logic Analyzer	37
20	Selecting a Wire to Trace With ELA	38
21	Selecting Additional Wires and Specifying ELA Trigger Condition	39
22	Collected ELA Results	39
23	Overview of Source Level Debugger	43
24	Levels of Abstraction Used in the Debug Database	44
25	Screen Shot of the Source Level Debugger	45
26	CPU Control Viewer Showing the CPU in the Middle of an Add Instruction	47
27	Assembly Source Code View Showing Breakpoints and Current Line of Execution	48
28	C Source Code View Showing Breakpoints and Current Line of Execution	48

29 Prototype Debug Environment. Note that the CPU control window shows the currently executing instruction, while the source code windows reflect the instruction currently being decoded. 49

30 Example Ptolemy Model 50

31 Output Ptolemy Sequence Ploter Using DSM 50

32 The Graphical User Interface for the Edit Distance Application 53

33 The JHDL Documentation Page 54

34 The JHDL API Documentation Page 56

35 The Board Model With Numerous Contexts Ready for Simulation 58

36 A Sample Circuit Loaded Into the Board Model 59

37 The View of the Sample Circuit After We Have Pushed Into ”testcircuit” 59

38 The Inside View of the Registers 60

39 The High Level View of the Scan-Instrumented Circuit 61

40 The ”testcircuit” Module After It Has Been Instrumented with Scan Chain 61

41 Muxes Are Added to each Flip-Flop 62

List of Tables

Table		Page
1	Summary of Files Used for Creating Readback Symbol Tables	17
2	Average Execution Time For Stages of a Hardware context-switch	25

1 Introduction and Motivation

This is the final report for the DARPA-funded project titled Unified Debug Environment for Adaptive Computing Systems. This work was funded through the DARPA Adaptive Computing Systems (ACS) program between 1999 and 2003 under U.S. Air Force Contract F33615-99-C-1502.

1.1 Research Motivation

For the 5 years prior to the start of this research, the BYU Configurable Computing Laboratory had been actively developing applications on a wide variety of adaptive computing systems, including: Splash2, Teramac, Ripp-10, WildForce, ClayFUn, and a variety of custom-constructed boards. The applications included general image processing, neural networks, image morphology, automated target recognition, and sonar beam forming. Although the group had been successful in this work, we noted the significant effort and patience required to complete each application. Based on that experience, we recognized the need for more mature verification tools and aids to assist the debugging process of FPGA-based designs.

The development of such debugging aids was the focus of this research project. The overall goal was to create a comprehensive debugging environment that mirrored –as much as possible– the symbolic debuggers that are used today by programmers developing programs in high-level languages. Instead of debugging designs primarily through simulation, designers using the resulting system would be able to debug their running applications directly on ACS platforms. The debug environment was intended to allow designers to:

- debug applications in the context of the original design environment used to create them,
- symbolically access the state of their running applications in real time using the original designs names for components and signals,
- transparently use ACS circuitry to monitor and control itself for debugging and analysis purposes.

In addition, the environment was intended to be multitasked, capable of remote execution, and able to operate offline, making it feasible for several designers working from different locations to access and develop applications on an ACS platform.

1.1.1 State of the Art: 1999

In 1999, we noted two reasons why ACS platforms were hard to program. First, the process of "programming" an ACS platform was often more akin to low-level circuit design than programming, at least as the process is understood by a typical programmer. Second and nearly as important, validating and tuning ACS applications was extremely difficult because ACS platforms generally suffered from a complete dearth of the kind of debug and analysis tools that the typical programmer has learned to take for granted. Tools such as symbolic debuggers, profilers, etc., are considered to be mandatory by the general programming community yet such tools were generally nonexistent in the ACS community. When the inherent programming challenge was combined with a lack of a comprehensive debug environment, mapping an application to an ACS platform was very challenging indeed.

ACS developers traditionally have had only one debugging tool available to them: simulation. However, it is simply not practical to completely debug and develop complex ACS applications using only simulators running on general-purpose processors. At best, current simulator technology can usually only provide simple debug capability in the early stages of development.

In 1999 the lack of debugging tools for available ACS platforms was puzzling to us as most of the early research ACS platforms all reported rudimentary debugging capability. Splash2 and DecPerle-1 could use readback to access internal FPGA state and could match this state with the symbolic signal names of signals found in the original design specification. Teramac and DecPerle-1 both had rudimentary breakpoint capability that allowed the designer to define a single hardware event that could be used to stop the global system clock. However, none of the platforms of 1999 that the authors were aware of provided debugging tools with the same primitive level of debugging capability as the early research platforms.

1.1.2 What Was Possible with 1999 Technology

Beyond the simple readback capabilities of Splash2 and DecPerle-1, we noted that there was a wide range of debug and analysis capability that could be implemented by using ACS circuitry to monitor itself, providing real-time hooks for debug and analysis. Just as software development environments add additional source code and symbolic information within a debug executable, an ACS application may include extra circuitry to augment visibility and other debugging aids. Because of the reconfigurability of ACS we proposed that such circuitry could be added as necessary during the development process, and then, just as easily removed or modified as required. Example debugging circuits envisioned included the following:

- ROUTE INTERNAL SIGNALS TO EXTERNAL I/O. Interconnect resources can be used to route an internal signal of interest to the external I/O of the device.
- REAL-TIME CIRCUIT PROBES. Unused ACS circuitry can be used to implement "probe" circuits that sample and store circuit activity at user-designated points during circuit execution.
- HARDWARE BREAKPOINTS. Unused ACS hardware can be used to implement complex, real-time hardware breakpoints.
- STATISTICAL MEASUREMENTS. Additional circuitry can be added to an ACS application for statistical measurements. Simple event counters or histogram generators may provide valuable insight into the operation of an external interface or processor.

Although all of the capabilities listed above are powerful and can be put to good use during ACS application development, almost none of them were used in practice in 1999 because of the *lack of a comprehensive debug environment.*

1.2 Overview of Proposed Research

From the original proposal to this project:

> We propose to develop tools and software for ACS that provide a unified, symbolic simulation and debugging environment, analogous to the debuggers that are commonly used to develop software with C or Java.

The proposal then went on to describe the specific areas where contributions would be made:

- UNIFIED DEBUGGING ENVIRONMENT. This effort would produce a unified, GUI-based browser where designers could graphically select hierarchical components, wires and other entities for examination, all within the context of the original design. One unified browser was to be developed for both simulation and execution. When running a simulation the browser would control the simulator and access simulation models; however, during execution, the tool would automatically find and access the circuit state on the physical ACS hardware in a way that was transparent to the user.
- MULTITASKING. The proposed debugging tools would automatically multitask making it possible for several users to simultaneously develop applications on a single ACS platform via the browser and other related software.
- REMOTE DEBUGGING. Both the browser and underlying software developed under this project would transparently support remote debugging, control and inspection. For example, users would be able to download a browser remotely as an applet which would connect with, and be used to control remote ACS hardware over the internet.
- AUTOMATIC DEBUG CIRCUITRY SYNTHESIS. Custom debug circuitry would be automatically synthesized and used to access internal system state transparently, using the configurable circuitry to monitor itself.
- CHECKPOINTING. This debugging software would support checkpointing: periodically saving the state of a simulation/execution run so that it could be restarted from that point at a later time.
- EXTERNAL HIGH-LEVEL SUPPORT. The debug system would support the debug of designs created using external high-level design tools. This would include the ability to convert design data from those tools into a format which the debugging system could manage.

2 Report Overview

Research into "A Unified Debug Environment for Adaptive Computing Systems" was conducted in accordance with the project's statement of work. The following sections document the results of that work.

The Unified Debug Environment (UDE) was built on top of the JHDL design system. JHDL is a Java-based design and debug system for configurable computing, the design portion of which was created in other research prior to this effort.

JHDL is freely available on the WWW at http://www.jhdl.org, under an open source style of license. Because of this, much of the documentation for JHDL (and thus the debug features created as a part of this research) exists at http://www.jhdl.org as a part of the on-line documentation. In many of the sections which follow, references are made to portions of that documentation. In each such case, where feasible, the appropriate sections of the on-line documentation have been printed and included as a part of the relevant section.

The remainder of this report is organized as follows. Section 3 documents the functionality of the DSM by describing its high-level structure and the services it provides. It then describes its internal structure to the extent needed to motivate the descriptions of the services and API's which follow.

Section 4 describes the platform control interface specification, how the DSM communicates with hardware platforms. It then describes a platform-specific backend, describing the software layer required for a real platform to interface with the platform independent software interface.

Section 5 describes debug circuitry synthesis and documents the API used for this purpose. Section 6 documents the mechanism for data interchange between the DSM and external tools (to enable the debugging of externally-created designs). Section 7 documents how a high-level tool works with the DSM through an API and then describes using a specific high-level tool with the DSM.

Section 8 documents the use of the debugging environment on selected applications, pointing out lessons learned and providing suggestions for further development of the debug system. Section 9 describes the on-line documentation (at http://www.jhdl.org for the Unified Debug Environment as embodied in the JHDL tool. Section 10 then documents the JHDL and debug support for the Sanders Context Switching Reconfigurable Computer (CSRC) context-switching device.

Finally, one appendix is attached. The appendix provides a summary of the various API's and their functions.

3 The Debug Services Manager (DSM)

The focus of this research project was to create a comprehensive debugging environment to exploit the unique capabilities of ACS, based around the concept of a Debug Services Module or DSM. The organization of the DSM is shown in Figure 1. There are three major sets of components that

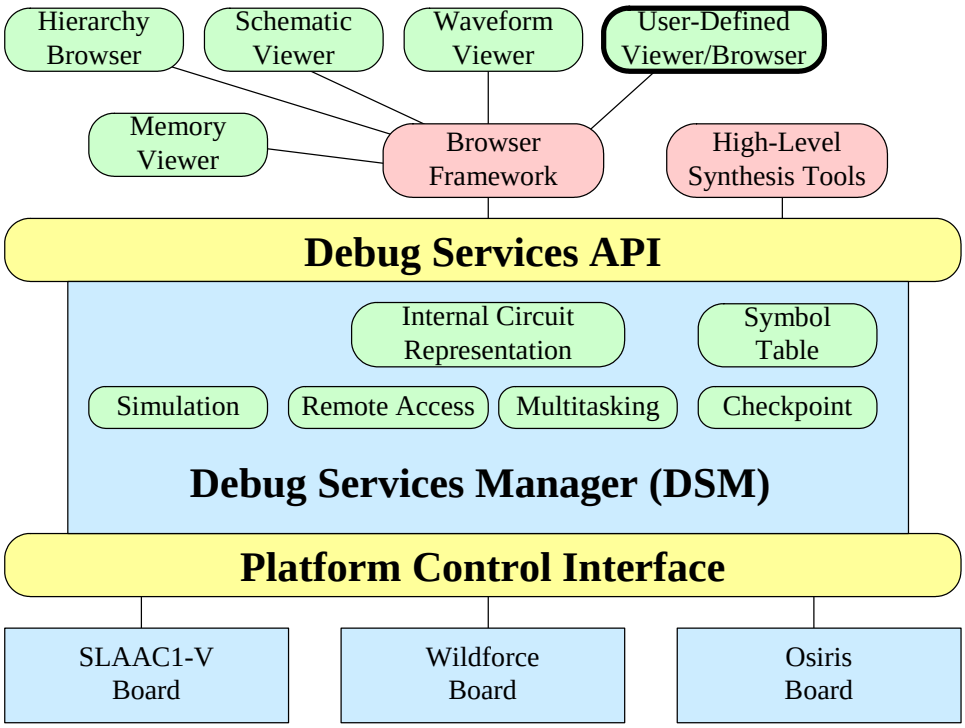


Figure 1: Organization of Debug Environment

comprise the system shown in the figure. At the top are the client programs that use and access the DSM, the browser, synthesis tools, etc. The DSM is shown in the middle, sandwiched between the Debug Services API at the top, which is the interface to the CAD tools, and the Platform Control Interface at the bottom, which is the interface to the ACS hardware. The low-level drivers to the ACS platforms are shown at the bottom with their corresponding interfaces to the DSM. What follows is a discussion of what functionality is provided by each of the DSM, the unified browser, and the low-level interfaces to the ACS platforms.

3.1 The DSM

The DSM is the software module responsible for providing debug and simulation services to other CAD tools such as the unified browser (which will be discussed later), synthesis tools, etc.

DSM Interfaces The DSM communicates with external CAD tools and ACS platforms through two standard interfaces: the debug services API and the platform control interface, respectively. As shown in Figure 1, all DSM facilities are accessed by the browser and other tools through the debug services API. The debug services API consist of a set of routines that can be invoked, for example,

to access internal DSM data, to single-step the application, etc. The DSM communicates with, and controls the ACS platform through the platform interface (shown at the bottom of the figure). The goal of the platform control interface is not to replace existing runtime software provided by the vendor but to provide a portable interface that isolates the DSM from all of the different, non-standard ACS drivers that currently exist. This makes it much easier to retarget the DSM to new ACS platforms as they become available.

DSM Core Functionality As shown in Figure 1, the DSM will provide core functionality for multitasking, checkpointing, remote access, monitoring, symbol-mapping, and simulation.

- INTERNAL CIRCUIT REPRESENTATION. The DSM maintains an internal, structural representation of the circuitry currently loaded for both simulation and execution. This internal representation is used for simulation, symbol mapping and other debug functions.
- SYMBOL-MAPPING. The DSM maintains a mapping of all designer signals from the internal structural representation down to the physical implementation so that all internal ACS signal values can be retrieved via their original designer-specified symbol names. This capability is essential to creating the ability to debug applications within the context of their original design.
- SIMULATION. The DSM provides a functional simulation capability so that ACS applications can be simulated and debugged offline. The simulator is also fully integrated with other functionality in the DSM to provide a unified simulation/execution environment whose full capabilities will be discussed later.
- REMOTE ACCESS. The DSM provides transparent, remote access to the ACS platform so that application development, debug and analysis can be performed remotely, say, across the Internet. This will be discussed in more detail later.
- MULTITASKING. The DSM provides a multitasked run-time environment that allows multiple designers to transparently access and use a single ACS platform for development and debug. This will be discussed in more detail later.
- CHECKPOINTING. The DSM provides support for automatically checkpointing a design, i.e., saving all of the current state of an ACS application such that it can be restarted at some later time. This capability is useful so that the results of long runs are not lost in the event of a power outage or computer crash. It also forms the foundation upon which Multitasking is built.

3.2 The Unified Browser

The unified browser is the primary interface to the DSM functionality and is represented by a subset of the boxes at the top of Figure 1 (Hierarchy Browser, Schematic Viewer, Waveform Viewer, Memory Viewer, and Browser Framework). Taken together, these pieces form a conventional design debug environment - they allow the user to view his circuit design in a number of forms and they allow the user to control the simulation or execution of the design.

The Unified Browser provides two forms of interaction to the debugging API: graphics-based interaction based on buttons, menus and circuit graphics and a command-line interface (CLI) that can be scripted for complex system monitoring functions and browser configuration.

The graphical interface consists of several panes: a circuit pane that can be used to hierarchically navigate the circuit, a waveform window that can be used to view the state of various signals during execution or simulation, a window for viewing schematics of the circuit, etc. Various buttons and menus are also provided for debugging and interacting with the circuit. A typical screen shot from the use of the unified browswer is shown in Figure 2.

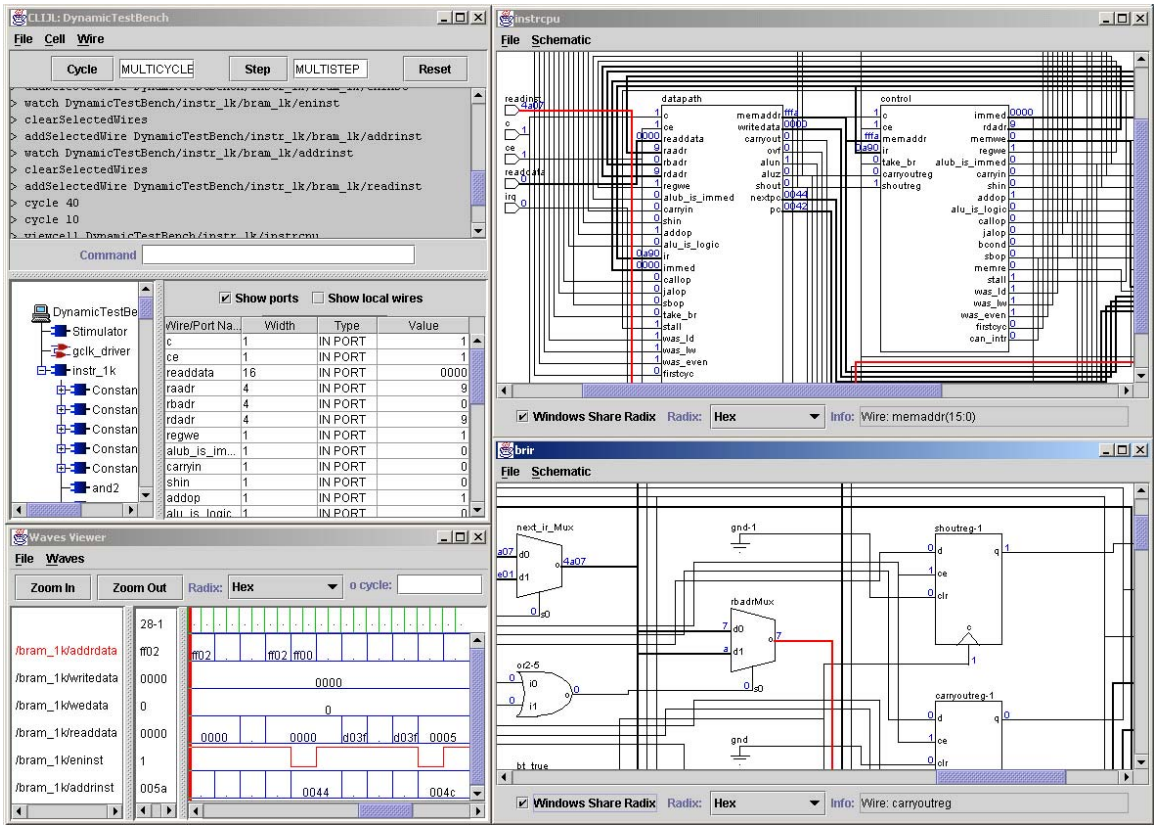


Figure 2: Unified Browswer Screen Shot

The CLI provides a scriptable command interface that can be used to perform complex debugging and system monitoring tasks. Complicated sequences of browser operations can be written as a script and then executed via the CLI. As will be discussed later, the browser was developed in a modular fashion so that various software components, e.g., the schematic generator, are available as stand-alone components and can be incorporated into other tools. The organization of the base JHDL system, unified browser, etc. is outlined in a number of publications: [16, 10, 9].

3.3 Unique Capabilities of the Debug Environment

The DSM supports a *unified* simulation/execution environment where designers are able to debug their designs either through simulation, execution or some combination of the two, using the same

user interface. *This is one of the key benefits of the DSM*. A unified environment such as this supports a single user interface that allows designers to request either simulation or execution, using the same exact set of commands for both. For example, within this unified environment, commands such as examine-variable, single-step, etc., are the same whether performing simulation or execution. This is a big advantage for designers because they can learn a single debugging environment that works for both simulation and execution –in contrast with current systems where execution and simulation environments are distinct and very different. Perhaps more importantly, the unification of simulation and execution leads to powerful new debugging strategies which were not feasible before. The following is a discussion of the some of the unique capabilities of the DSM that are due to its integration of the capabilities shown in Figure 1.

Checkpointing and Simulation Checkpoint data from hardware execution can be used to initialize the simulator. This makes it possible to start the simulator from any saved checkpoint of some prior hardware run. Consider the following scenario. The designer runs an application for a few seconds on the ACS platform (e.g., 120,000,000 clock cycles worth), with a checkpoint occurring every 1,000 clock cycles. While analyzing the execution data, the designer notes something interesting occurring at the 100,000th cycle and decides to investigate further. The designer might wish then to restart the application, starting from the data checkpointed at the 100,000th clock cycle. Now, under the DSM the designer has two choices: either simulate the application, starting from the checkpointed data, or restart the application on the ACS platform, again using the checkpointed data. If the ACS platform is unavailable or if offline simulation is more desirable for some other reason, the designer can start the simulation up from the checkpointed data and simulate forward to the event of interest. In practice, if data were checkpointed every 1000 clock cycles as in this example, offline simulation performance would usually be sufficient for debugging between checkpoints. By combining simulation with execution in this manner, simulation becomes a much more powerful tool that can be used to offload the ACS platform during debugging. This capability can also be used transparently, via the multitasking capability of the DSM. For example, assume, as in the previous example, that a run of 120,000,000 clock cycles was checkpointed every 1000 clock cycles. The designer then requests to restart the application (execution) at the 100,000th clock cycle. The DSM can then choose whether to perform the designer's request via execution or simulation. For example, there may not be a time slice available on the ACS platform and in this case the DSM may satisfy the request with simulation. In this scenario, simulating the request may be faster than executing it (lower latency) because the designer's request can be immediately fulfilled.

Remote Access and Multitasking Using the checkpointing facility just describe, the DSM provides the ability to multi-task or share an ACS platform. In addition, it provides for remote access of ACS platforms. Both of these capabilities are described in detail later.

Circuit Monitoring and Symbol Mapping Combining circuit monitoring and symbol mapping allows for advanced circuit monitoring techniques to be used in a user-friendly way, within the context of the original design. For example, a designer may request a "watch" on some signal in the design. At this time, the designer could also specify that the watch needed to be performed in real time. Given that the monitoring circuitry had already been embedded in the design, the DSM could access the signal values and display them using the same names as they were specified in the original design. This makes the monitoring circuitry much easier to use.

In summary, it can be seen that many of the capabilities of the proposed debug environment are due to the careful integration of a simulator and a richly-featured execution environment as detailed above.

3.4 Internal Circuit Representation

Prior to the commencement of this research project, a version of the JHDL design tool was already in use[9, 10]. The DSM and Unified Browser were built on top of that foundation. Although a full discussion of JHDL as a design tool is beyond the scope of this report the following is provided as introductory material to better understand the research results.

JHDL provides a library of Java classes which the user extends to build his circuit. A sample circuit for a full adder in shown in Figure 3. The JHDL description for this circuit is shown in Program 3.1. After first importing JHDL's libraries (lines 1-3) the user's design is declared as a Java class (lines 5-6). Declaring the user's design to extend the JHDL built-in *Logic* class, makes it inherit numerous data structures and functions from that class (ports, a place in the circuit structure hierarchy, access to functions for creating wires and logic, etc).

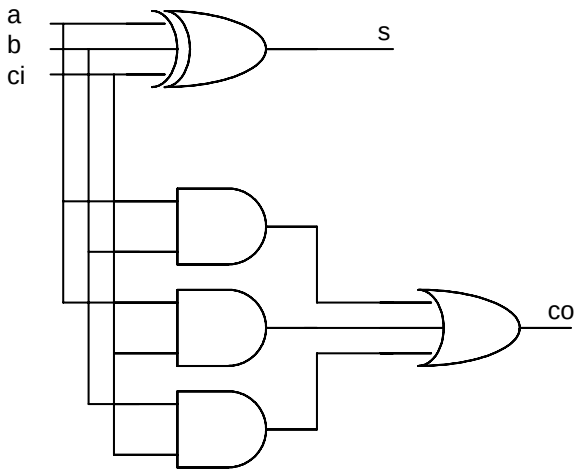


Figure 3: Full-Adder Example Circuit

In lines 8-15 the design's ports are declared. Since the user's design (a full adder in this case) is a normal Java class, it needs a constructor routine which is called when a full adder instance is desired in a design. The constructor function is defined in lines 17-38. The first parameter passed to the constructor call is a reference to the circuit element which will be the parent of the newly constructed full adder in the circuit hierarchy. The remaining parameters are references to the wires in the parent circuit element which will be connected up to the ports of the full adder. In lines 26-32 the formal binding of those wires to the proper ports from the full adder's CellInterface are made using ***connect()*** calls. Finally, in lines 34-36 the actual logic making up the full adder is created. The subroutines used to build the circuit (***and()*** for example) are methods inherited from the built-in JHDL class named "Logic".

Although the language the FullAdder description is written in looks like an HDL, it is not an HDL. It is simply conventional Java; no extensions or modifications to the language were made. As such it

Program 3.1 Full-Adder Example

```
( 1)  // Import the base libraries for JHDL design
( 2)  import byucc.jhdl.base.*;
( 3)  import byucc.jhdl.Logic.*;
( 4)
( 5)  // Our FullAdder is a Java class.  Begin the class declaration here
( 6)  public class FullAdder extends Logic {
( 7)
( 8)    // This is cell's interface (ports)
( 9)    public static CellInterface[] cell_interface = {
(10)      in("a", 1),
(11)      in("b", 1),
(12)      in("cin", 1),
(13)      out("sum", 1),
(14)      out("cout", 1)
(15)    };
(16)
(17)    // This is the constructor - it gets called when a new FullAdder is desired
(18)    public FullAdder(Node parent, Wire a, Wire b,
(19)                     Wire cin, Wire sum, Wire cout) {
(20)
(21)      // Since we extend Logic, always have to call its constructor
(22)      // as the first thing.
(23)      super(parent);
(24)
(25)
(26)      // Connect the wires passed in as parameters to the constructor to
(27)      // the ports from the CellInterface above.
(28)      connect("a", a);
(29)      connect("b", b);
(30)      connect("cin", cin);
(31)      connect("sum", sum);
(32)      connect("cout", cout);
(33)
(34)      // Build our logic as a collection of gates.
(35)      or_o( and(a,b), and(a,cin), and(b,cin), cout ); /* cout is output */
(36)      xor_o( a, b, cin, sum );                        /* sum is output */
(37)
(38)    }
(39)  }
```

can be compiled with any java compiler and the resulting class file executed like any other Java class file. The key feature of the circuit description is that executing the resulting class file results in the *contstruction* of a data structure representing the circuit. For example, since the FullAdder design shown in Program 3.1 extends class "Logic" (which extends "Cell" which extends "Structural" which extends "Node" - all transparently to the designer), the by-product of calling FullAdder's constructor is to create the data structures necessary to represent the FullAdder, its ports and wire connections, and its constituent subcells (OR, AND, and XOR gates). This data structure is the "Internal Circuit Representation" shown in Figure 1.

In actuality, JHDL builds this structure using the build stack which allows the circuit to be built in any order (top-down, bottom-up, etc). As long as an circuit is not finished, it can be added to or modified at any level. Elements such as the FullAdder described above are generically referred to as *cells* in the data structure. Cells are either terminal and contain behavior or they are hierarchical and contain child cells. The FullAdder above is hierarchical. Both hierarchical cells and wires include the ability to express heirarchical relationships and connectivity. In addition, wires may be aliased, bundled, and recombined. These abilities increase the ease with which a design may be expressed and aid in the partioning of design structures.

In addition to reflecting circuit structure, the JHDL data structure contains information which can be used to simulate the circuit. Hierarchical cells are simulated simply by simulating their children's behavior. Non-hierarchical cells (terminal cells) have no children and represent leaf cells in the design hierarchy (the primitive building blocks provided by the technology library such as gates and flip flops). JHDL provides libraries of such cells for the technologies it supports (Xilinx 4K, Xilinx Virtex, Xilinx Virtex2, CSRC, and Altera). Such cells contain behavioral descriptions as a part of their definitions. The circuit data structure provides for the storage of state associated with terminal cells and wires such that simulation is easily done by executing the behavior of terminal cells and propagating changing results between terminal cells along wire pathways in the proper sequence.

Users can also define behavior for custom cells which they create. In addition, users can define behavior or for any level of heirarchy that exists in the design. The hierachical definition of behavior aids in rapid application development. Models of high-level behavior may be created first and the underlying design details can be generated after the high-level behavior has been simulated and determined to be correct.

In summary, the components used to build a design (cells and wires) are simulatable and no translation is necessary to start simulation. Rather, simply executing the Java class files representing the user design builds the necessary circuit data structures in preparation for simulation. The DSM which was the subject of this research project relies heavily on the resulting "Internal Circuit Representation" to provide the services it provides.

3.5 The Simulator

The DSM contains a simulator which uses the features of the circuit data structures described above to provide conventional logic simulation. In preparation for simulation and to provide efficient simulation, a levelization or scheduling of the circuit is done. After the initial circuit scheduling, simulation consists of repeated synchronous or propagate calls according to circuit element behavior and the circuit schedule. The simulator supports multiple clocks, as well as gated clock circuit designs. The simulation provided by the simulator can be used in several different debug activities.

As shown in Figure 4, all access to the simulator is brokered by the Hardware System (HWSystem). HWsystem is root node of every design and provides access to the functionality and debug capabilities of the Simulation System. Specifically, HWSystem provides access to externally update the simulator memory and state elements. HWSystem also provides access to controlling the simulation of designs. Feedback on the simulation may be requested through HWSystem as well. Finally, HWSystem allows access to the SimulatorCallBacks so simulation events can be tracked.

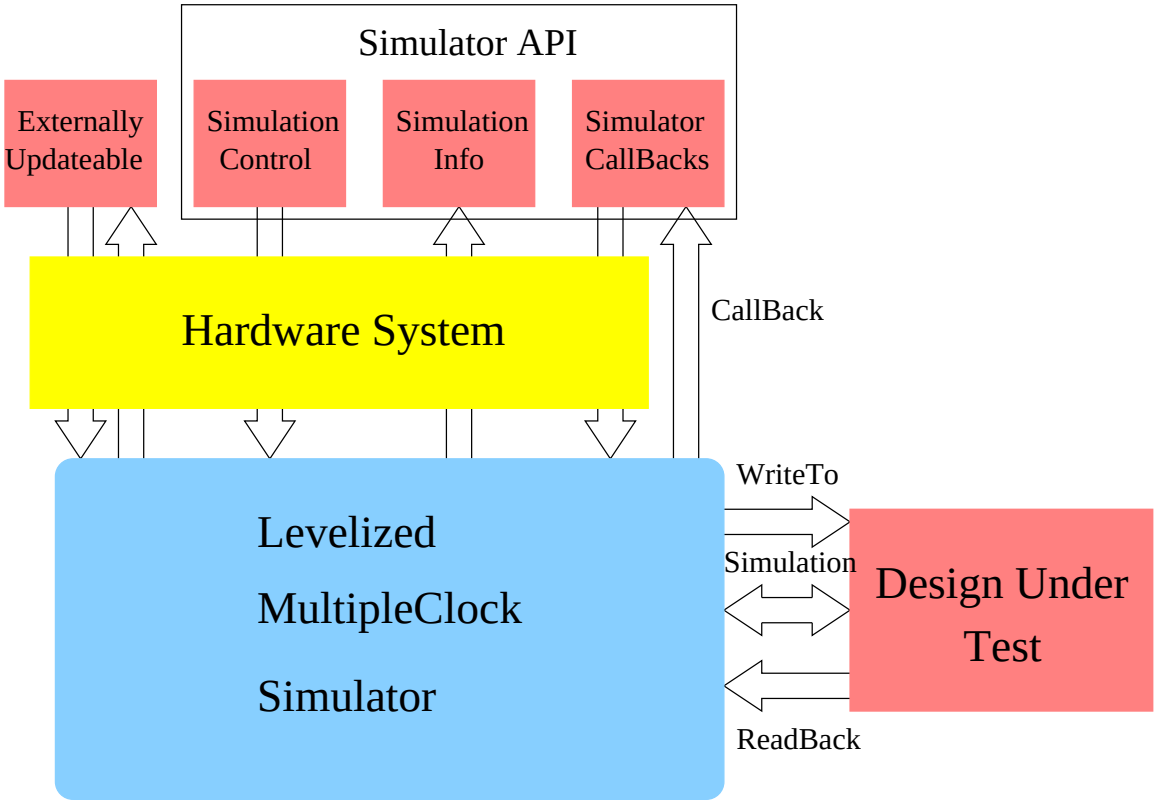


Figure 4: Interfaces Used In Conjunction With Simulation System

Externally Updateable API The simulator has the ability to have its state externally updated and extracted. This means that the state of all memory elements (e.g. registers, RAMs, ROMs, etc.) can be set to any value by processes outside of the simulator. These values can also be extracted and serialized to files or other storage. This provides the ability to checkpoint software simulation, as well as import the state of a hardware design into the original circuit representation.

An example, shown in Figure 5, of using the Externally Updatable API occurs during in the Hardware Mode of the HWSystem. In Hardware Mode, simulation of the circuit is pre-empted so that actual values retrieved from hardware may be shown in circuit schematics and other viewers. A board model, which controls operation of the reconfigurable hardware board, executes some number of cycles on the hardware. The state elements are extracted from the board itself and mapped to the state elements in the circuit representation in the simulator through the externally updateable interface. The extracted state is written into the circuit state elements and the simulator then resolves the state of all non-updateable elements (a common kind of non-updateable element would be a combinational logic gate - its output value is not retrieved from the executing hardware but can

be re-created once the extracted state is back-annotated into the simulation data structures). Board models are described in more detail in Section 4 and the Hardware to Circuit element mapping is described in Section 3.6.

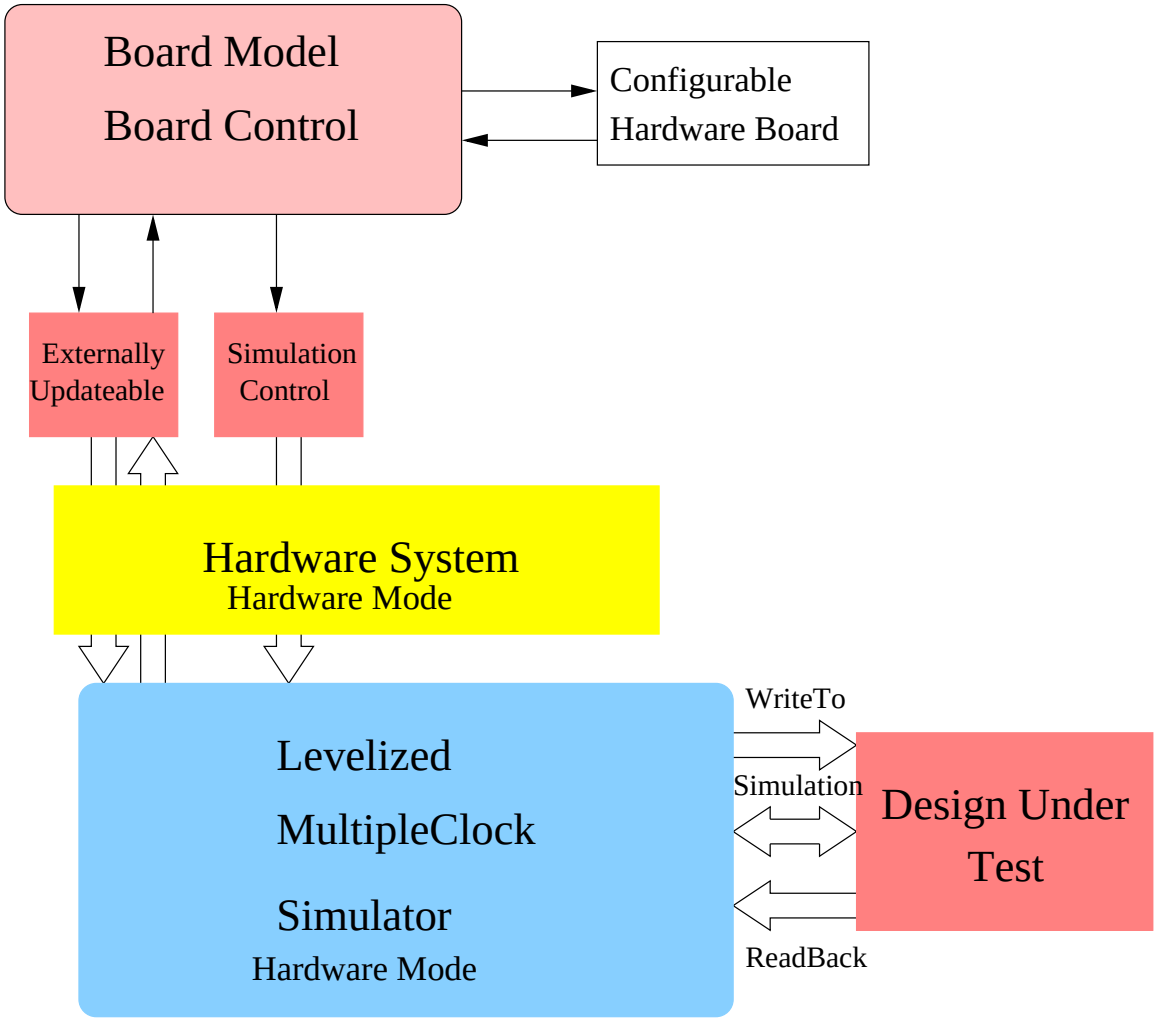


Figure 5: Simulation System Operating in Hardware Mode

The Externally Updateable API has only three calls as shown in Program 3.2. The ExternalUpdate-Manager (EUM) has several different functions. First, the EUM can read the state of a simulating circuit. The state can be written to a file or read from a file. Second, EUM has the necessary calls to extract state from a reconfigurable hardware board. Finally, the EUM can map the state from hardware to the simulation update elements. These functions cannot be controlled externally. The only external modification can be to the lists of ExternallyUpdateable Cells.

The modification of ExternallyUpdateable Cells, or in other words the reading and writing of memory elements, supports a wide range of debugging and control features in JHDL. The state of a digital circuit can be derived from the memory elements found in the circuit itself. Access to the memory elements in JHDL provides the ability to record the state of a circuit for later simulation or for later examination. The same access allows the simulator to be updated to reflect the current state of external hardware. In this way, designs executing on external hardware can be visualized in

Program 3.2 Externally Updateable API

```
/** Access the ExternalUpdateManager
  *  @return the external_update_manager
  */
 ExternalUpdateManager getExternalUpdateManager() {
   return external_update_manager;
 }

 /**
    Method to get the list of ExternallyUpdateableCells
    @return List of ExternallyUpdateableCells
 */
 public ExternallyUpdateable[] getExternallyUpdateableCells() {
   return external_update_manager.getExternallyUpdateableCells();
 }

 /**
    Method to get the list of all LargeExternallyUpdateableCells
    @return List of LargeExternallyUpdateableCells
  */
 public LargeExternallyUpdateable[] getLargeExternallyUpdateableCells() {
   return external_update_manager.getLargeExternallyUpdateableCells();
 }
```

circuits and in external plug-ins through the SimulatorCallBack interface.

Simulator API Direct control of simulation operations provides methods to start, stop, move to a given location of the simulation, etc. The direct control is identical when simulating a design or interacting with a design executing on a hardware platform. Direct control allows external elements interacting with the DSM to have cycle based control over simulation.

Direct control of the simulation operations is given by the ***cycle()***, ***step()***, ***skip()***, and ***haltSimulator()*** methods. As described in Program 3.3, these methods move the simulator forward a fixed number of cycles or steps and simulate the circuit elements in a design. ***haltSimulator()*** allows the simulator to be asked to execute a large number of cycles and then halt the simulation once a given condition is reached.

Other infomation can be queried from the simulator. The simulator keeps track of errors in simulation, clock events and cells that which were found, but were not scheduled. This information can be queried via HWSystem through various calls. The calls and the information returned are detailed in the HWSystem documentation, which can be found at:

http://www.jhdl.org/docs/docs/api/byucc/jhdl/base/HWSystem.html

The SimulatorCallback API The SimulatorCallback interface allows the registering of methods which are called in response to circuit resets, updates and refreshes. This allows the creation of Java programs which gain control between simulation cycles so they can follow the circuits' behavior. The SimulatorCallback interface is provided to assist in a variety of tasks including: (1) the writing

Program 3.3 Simulator API

```
/** This method is invoked to advance the simulator by
 *  no_clocks. This routine blocks until the simulator has finished
 *  the requested number of clock cycles.
 *  @param no_clocks the number of clocks to run the simulator.  */
public void cycle(int no_clocks);

/** This method is invoked to advance the simulator by no_steps.
 *  This routine blocks until the simulator has finished the
 *  requested number of clock steps.
 *  @param no_steps the number of steps to run the simulator */
public void step(int no_steps);

/** This method is invoked to advance the simulator by no_clocks.
 *  This routine blocks until the simulator has finished the
 *  requested number of clock cycles.  This differentiates itself
 *  from step in that the observers are not updated and makes it
 *  possible to have correlating commands in hardware and simulation
 *  modes.  */
public void skip(int no_steps);

/* sets a flag that causes the simulator to break out
   of it's run() loop
*/
public void haltSimulator();
```

of self-validating designs, (2) the creation of custom graphical browser elements, and (3) the creation of formatted simulation output. Any JHDL cell can implement this interface. If so, the cell must implement three methods as shown in Program 3.4. This interface allows external elements to

Program 3.4 The SimulatorCallback Interface API

```
// Called each time the simulator resets
public void simulatorReset();

// Called each clock step after the circuit settles
public void simulatorUpdate(int cycle, int phase);

// Called each time the JHDL system deems a refresh of all
// GUI elements is warranted (i.e. after a simulation run which
// could be of multiple cycles)
public void simulatorRefresh(int cycle, int phase);
```

"plug-in" to the operation of the simulator itself. External code that implements the SimulatorCallBack interface can be registered with the simulator. The external code will them be informed when the simulator is reset or has been asked to simulate. Simulator call backs are described in the JHDL documentation at:

http://www.jhdl.org/docs/docs/usersManual/simulatorcallback.html

3.6 Symbol Mapping Facility

The symbol mapping facility is a service provided by the DSM off-line. That is, the DSM provides facilities which can be used as a preprocessing step to determine a logical to physical mapping [3] for a given design. These are provided as platform dependent implementations; implementations for Xilinx XC4000 and Virtex FPGAs have been created[1]. The logical to physical mapping maps state elements in the JHDL circuit to locations in the FPGA readback bitstream as shown in Figure 6. This is done through the .rbentry file. This file contains a list of names of state elements and their corresponding offsets in the readback bitstream. It also contains other pieces of information which are used to properly interpret the state data from the readback bitstream.

The .rbentry file is used by the ReadBackManager in the DSM to extract the circuit state from the executing circuit and back annotating it into the simulation data structures using the ExternallyUpdateable interface described above. This is at the core of providing a unified environment for simulation and hardware execution.

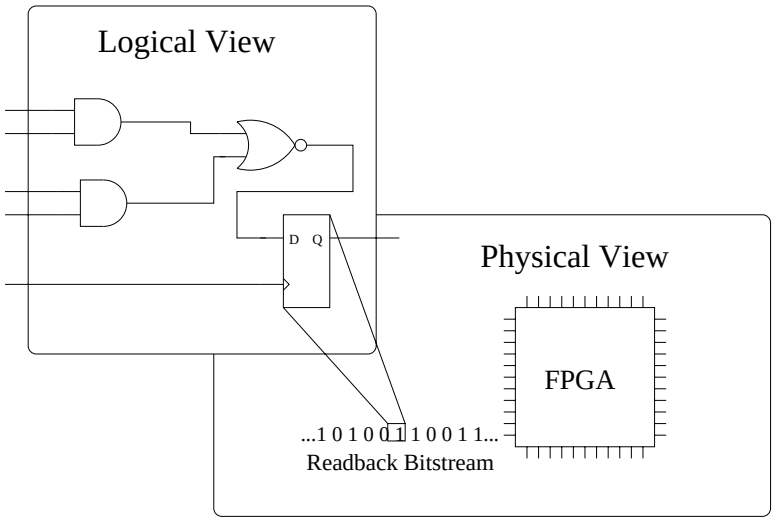


Figure 6: Logical to Physical Mapping

The process of creating the logical to physical mapping is quite involved and is described in detail in the dissertation "Logical Hardware Debuggers for FPGA-Based Systems" [4], and provides a new approach to making the mappings. In past efforts, tools were provided which provided a limited form of logical to physical mapping based on signal names. An example of this is the ***t2*** tool provided with the Splash-2 CCM[1]. This proved to be quite limiting, as signal names are merged and changed during the backend tool flow (during mapping, placement, and routing). The new approach developed here takes a cell oriented approach, which provides much better mappings. The rest of this section is an excerpt extracted from "Logical Hardware Debuggers for FPGA-Based Systems" [3] and [4] which explains the process of creating a logical to physical mapping.

Figure 7 depicts this process for creating hardware symbol tables for Xilinx XC4000 and Virtex designs created with JHDL. As the figure shows, XC4KToJHDLSyms[2] and VirtexToJHDLSyms[3]

[1]These implementations are found in the classes byucc.jhdl.platforms.util.readback.Xilinx.XC4000.XC4KToJHDLSyms and byucc.jhdl.platforms.util.readback.Xilinx.Virtex.VirtexToJHDLSyms.

[2]Java class: byucc.jhdl.platforms.util.readback.Xilinx.XC4000.XC4KToJHDLSyms

[3]Java class: byucc.jhdl.platforms.util.readback.Xilinx.Virtex.VirtexToJHDLSyms

are the programs implementing this process for JHDL designs based on Xilinx XC4000 and Virtex FPGAs, respectively. The process uses four files to build symbol tables: the .rbsym file, the Map Report file, the Xilinx Design Language (XDL) file, and the Logical Allocation file. The first of these identifies the logical state elements to locate when building the design's hardware symbol table and is created by JHDL at netlist time. The Xilinx tools generate the remaining three files. The Map Report, or .mrp, file describes how logical designs described in FPGA library primitives are mapped to FPGA resources, including how the design was optimized during the mapping process. The XDL file is a textual representation of how the FPGA's resources are configured to implement the design. As mentioned before, the Logical Allocation (.ll) file relates readback bitstream data to the state of FPGA resources. Table 1 summarizes the types and functions of files used in the process.

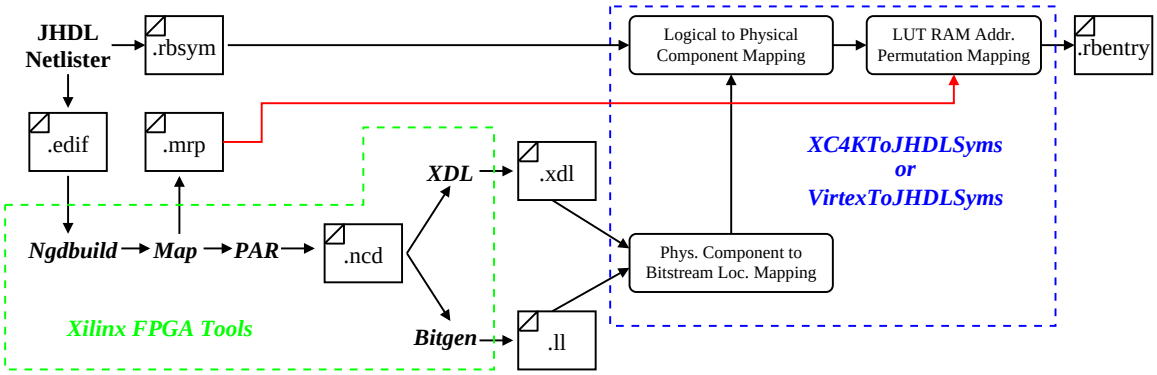


Figure 7: Process of Logical-to-Physical Mapping

Table 1: Summary of Files Used for Creating Readback Symbol Tables

Name	Creator	Description
.rbsym	JHDL	Netlist of logical state elements for readback
.ll	Xilinx, bitgen -l	Report listing the locations of sampled FPGA state in the readback bitstream
.mrp	Xilinx, map	Report describing a design's mapping to FPGA resources, including logic optimizations
.ncd	Xilinx, par	Binary, closed description of a design's FPGA physical implementation
.xdl	Xilinx, xdl	Textual, open description of a design's FPGA physical implementation
.rbentry	JHDL, XC4KToJHDLSyms JHDL, VirtexToJHDLSyms	A hardware symbol table relating logical design elements to locations in the readback bitstream

Conceptually, the process of creating the hardware symbol table involves three main steps. First, FPGA state elements (flip-flops and RAMs) found in the physical design described by the XDL file are associated with the locations of their state data found in the readback bitstream, as described by the .ll file. After this, the logical state elements found in the .rbsym file are correlated with the corresponding physical FPGA resources from the XDL file based on design instance names. This correlation effectively results in a map relating *logical* state elements with the locations of their sampled state found in readback bitstreams. The third step compares the ordering of physical address signals for LUT RAMs with the original ordering of the logical address signals to determine the address permutations performed by the Xilinx par software and then appropriately permutes the

readback bitstream locations in the hardware symbol table to reflect the permutation. To make this comparison between logical and physical signal names possible, the software determines how signal names are modified due to optimizations based on the information from the .mrp file, a fairly involved task.

The result of the entire process is an .rbentry file which is the hardware symbol table for readback. Unlike previous work, the hardware symbol tables generated by XC4KToJHDLSyms and VirtexToJHDLSyms map 100% of the state information provided by readback to flip-flop and RAMs in logical designs. Further, the process is much simpler to perform since correspondences between logical and physical state elements are made directly based on instance names rather than indirectly through output signal names.

3.7 Checkpointing

Checkpointing provides the ability to capture the complete state of the DSM and save this state for use at a later time. Any archived state of the DSM can be restored into the DSM for further debugging and testing. This is extremely useful when debugging long simulation runs or hardware execution tests. Checkpointing is also used to facilitate multitasking which is described in the next section.

The benefits of checkpointing can be seen by an example debugging session presented in Figure 8. In this example, the DSM is used to monitor the execution of a reconfigurable system on an ACS platform attached to a host computer. The user of this system executes this hardware platform every one hundred thousand clock cycles and evaluates the state of the system within the DSM. At each interval, the system state is stored to disk as an archived simulation checkpoint. After several checkpointing intervals, an error is identified in the system. In an effort to identify the cause of this problem, the user retrieves an old checkpoint and restores the state in the checkpoint into the DSM. At this point, the user can probe the system in more detail to identify the cause of the problem.

3.7.1 Checkpointing Architecture

Checkpointing is implemented using the object *serialization* capability built into Java. Object serialization allows the run-time state of Java objects to be encoded as a stream of bytes and stored into a file or other appropriate I/O object. Java serialization supports the complementary reconstruction of the run-time object state from the stored data stream. Java serialization can be used to store the relevant state of the DSM to a file.

The interaction between the DSM, serialized Java objects, and the hardware platform is shown in Figure 9. A checkpoint of the current state of the DSM can be made by serializing all the appropriate system state and storing the serialized byte stream onto a file. Later, this stored checkpoint file can be restored into the DSM by deserializing the byte stream into the DSM data structures. The updated state of the DSM reflects the exact state of the DSM at the point in which the checkpoint occured. Details of the JHDL checkpointing infrastructure can be found in Chapter 2 of [13].

The checkpointing facility integrated within the DSM can be used to checkpoint the state of hardware as well as the state of the software simulation. As shown in Figure 9, hardware checkpointing is implemented using the the Platform Control Interface (see Section 4). To checkpoint an ACS platform, the state of the hardware must first be updated within the DSM using readback and other

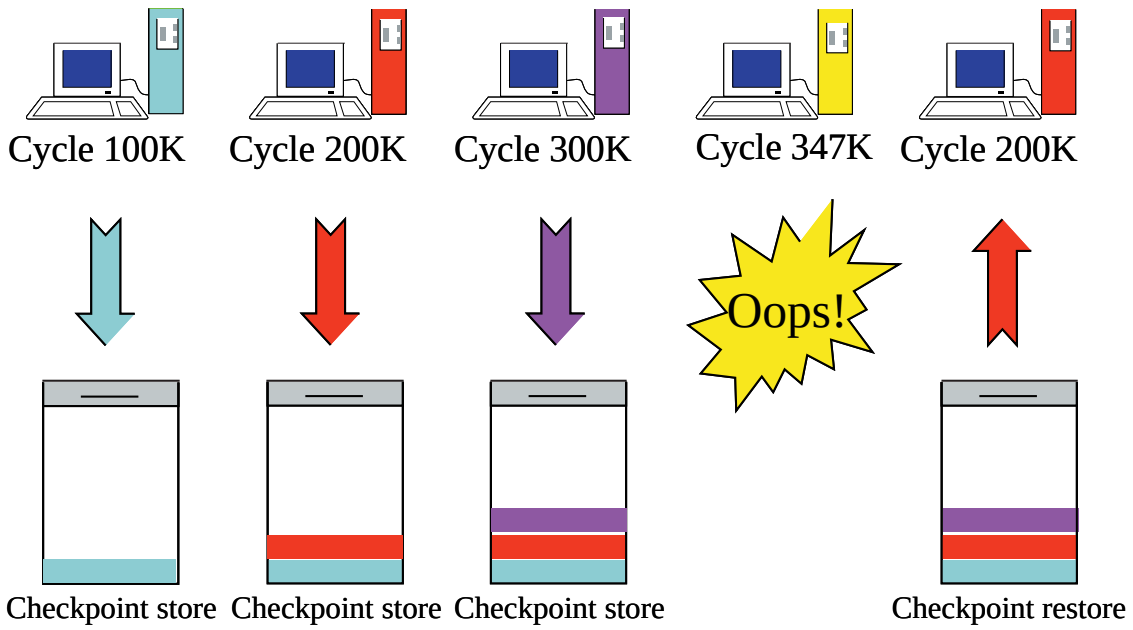


Figure 8: Example Use of Checkpointing

platform control interfaces. Once updated within the DSM, the hardware state can be stored to disk using the Java serialization API.

To fully support checkpointing in hardware, the DSM must have the ability to *restore* the state of the hardware. The process of restoring hardware state may be supported by the platform control interface through *writeback*. If writeback is available for a given platform, the checkpointing facility has the ability to move system state seemlessly between software simulation, the hardware platform, and disk storage. This allows the state of a simulation to be saved and transferred to the hardware for faster execution. Alternatively, a system checkpoint can be moved from a hardware platform to a simulation environment for greater visibility. Such state checkpoints can be saved to disk and archived for further analysis at a later time.

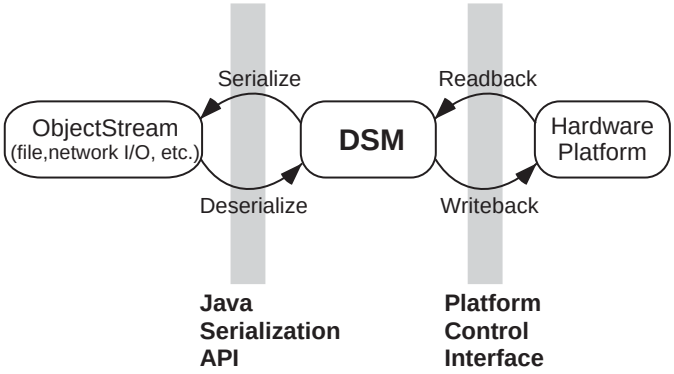


Figure 9: Checkpointing the DSM

3.7.2 Checkpointing API

The checkpointing API is found exclusively within the JHDL byucc.jhdl.base.HWSystem class. These methods can be accessed directly within a testbench or through an appropriate board support package or GUI interface such as CVT. The checkpointing API supports four major operations: save state, read state, writeback, and readback. These methods are shown in Listing 3.5.

Program 3.5 Checkpointing API

```
/** These methods cause the HWSystem to write the state of all
 *  ExternallyUpdateable and Checkpointable objects into the appropriate
 *  I/O object. */
public writeSystemState();
public writeSystemState(String filename);
public writeSystemState(OutputStream os);


/** The methods listed below are used to update the system state
 *  with a checkpoint. **/
public readSystemState(int cycle, int step);
public readSystemState(String name, int cycle, int step);
public readSystemState(InputStream is);

/** This method is used to write the current state of the simulator
 *  to the hardware. This is will only work if the current loaded
 *  model will allow it. **/
public writeSystemStateToHardware();

/** This method will perform hardware readback without stepping the
 * hardware clock. **/
public updateHardware();
public cycleHardware(int steps);
```

Save State Several methods are available for storing the state of the DSM as a checkpoint. These methods, all variations of writeSystemState, serialize the relevant data within the DSM and store them to disk or to a OutputStream.

Read State Several methods are also available for restoring the state of a checkpoint into the DSM. These methods take a saved checkpoint and restore it into the DSM. In some variations of this method, the cycle of the checkpoint is given to set the cycle number of the simulator.

Writeback The method writeSystemStateToHardware is provided to perform a writeback on the hardware platform. This method captures the current state of the DSM and sets the state of the hardware platform to reflect that of the DSM simulator. Note that this method is only avaliable for platforms that provide support for writeback.

Readback The methods updateHardware and cycleHardware are used to capture the state of the hardware and set the state of the DSM. This involves readback of the programmable logic resources

and non-programmable platform resources such as memories and resources. Again, these functions are only available on platforms that support reading hardware state.

3.7.3 Checkpointable and ExternallyUpdatable Interface

The checkpointing facilities do not save all of the state of the DSM. Instead, checkpointing limits data within the checkpoint store to the state of programmable logic resources and other platform specific data. Two interfaces are available to tag hardware resources for checkpointing: ExternallyUpdateable and Checkpointable. Each of these interfaces will be described below.

ExternallyUpdatable The ExternallyUpdatable interface is used to tag simulation primitives whose state can be set through readback. This interface is implemented by sequential primitives such as flip-flops, LUT RAMs, and block rams. This interface requires each such primitive to implement the updateState method. This method is called during a readback to update the state of the primitive in the DSM simulation environment.

Checkpointable The Checkpointable interface is used to tag other JHDL objects for checkpointing. In many run-time environments, the state of non-primitive objects must be saved during a checkpoint. These objects may included off-chip memories, testbench values, and other application-specific parameters.

3.8 Multitasking

Hardware multitasking provides a run-time environment that allows multiple designers to transparently access and use a single ACS platform for development and debugging. Much like a modern multitasking operating system, hardware multitasking increases the utilization of an ACS platform by arbitrating the use of the hardware systems for multiple simultaneous users. This multitasking environment schedules user execution in a manner that maximizes the utilization of the ACS system. Additional details describing multitasking can be found in [13].

Figure 10 demonstrates how hardware multitasking can be used to support multiple users. In this example, four users are connected to the ACS hardware simultaneously. Each user is working on a different ACS application and requires varying levels of access to the ACS platform. One developer may be executing an application for a long period of time and require exclusive use of the board. Other users may be accessing the board occasionally to test sections of the hardware or single-step through a critical time sequence. If one user is allowed to monopolize the hardware, other users may not be able to perform the periodic debugging necessary. Multitasking is provided to maximize the availability of the hardware for multiple users by supporting automated *context-switching* of hardware resources.

3.8.1 Hardware Context Switching

The key operation performed by the multitasking system is a hardware context switch. In a hardware context switch, the currently executing hardware must be swapped with the hardware of some

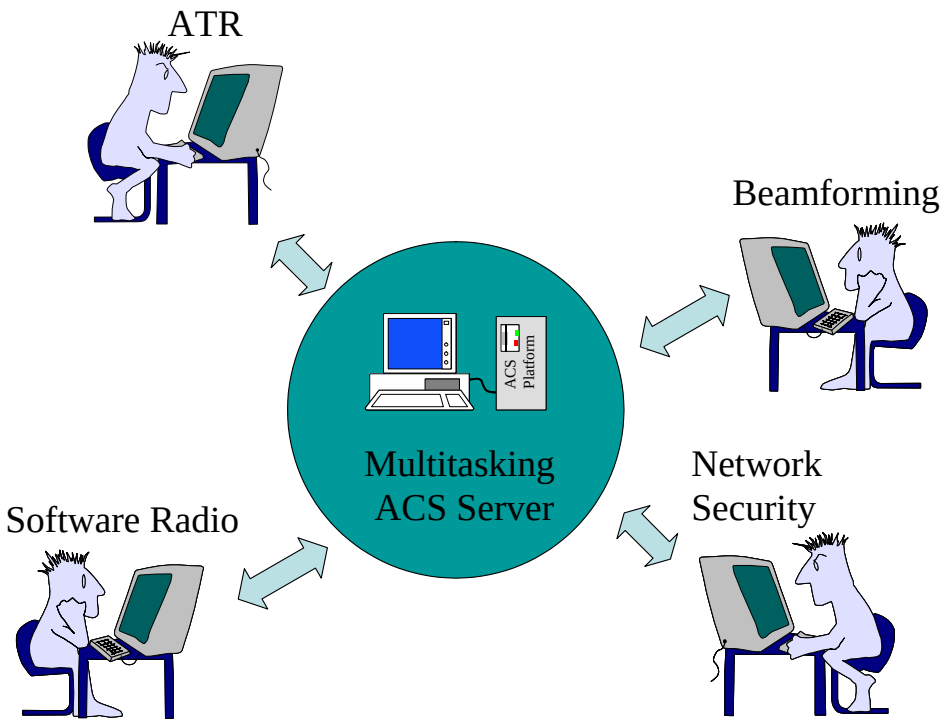


Figure 10: Multitasking Three Applications

other "stalled" hardware context. The multitasking system must perform this switch without any unintended side effects – the new process must be restored to the hardware as if it had never stopped. Further, the outgoing process must be stored in a manner that allows it to be properly restored at a later time.

A hardware context switch is performed using four important steps: stop hardware, capture hardware state, restore hardware process, and restart hardware. This four step process is depicted in Figure 11 for a context switch from Process A to Process B. Details of these steps are described below.

1. **Stop Hardware** The first step of a context-switch is to *stop* the currently executing hardware process. To support context-switching, the owner of the resource must have the ability to stop the process in the middle of its execution sequence. In a configurable hardware system, there are many concurrently operating circuits that all must be stopped *simultaneously*. All of the FPGAs must be stopped at the same time to ensure that the entire system has been halted in a known and restorable state. The most straightforward method of simultaneously stopping synchronous hardware is to stop the global clock. When the global synchronous clock has been stopped, the configurable hardware will no longer modify its state registers or memory.

2. **Capture Hardware State** The second step is to capture the state of the hardware using the readback and symbol mapping techniques described earlier. An internal checkpoint of this state is made in preparation of a future reloading of the hardware context. Unlike software multitasking, hardware systems contain a large amount of state that is difficult to access. The state associated with a configurable application includes the design configuration, every

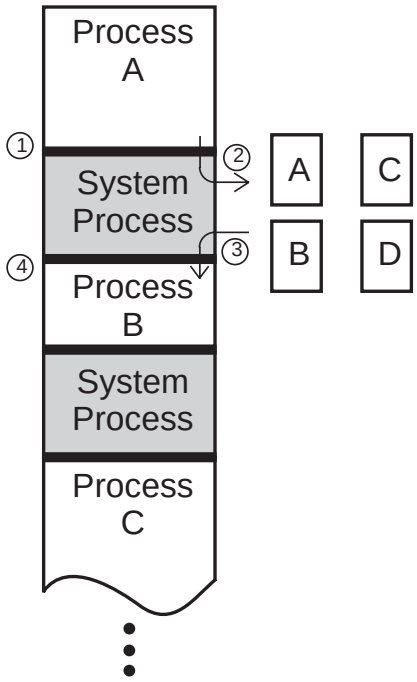


Figure 11: Four Steps of a Hardware Context Switch

flip-flop and memory element within its FPGAs, and the contents of any external memories attached to the programmable logic. This large amount of state requires memory space for state storage and a relatively long time to access the state.

3. **Restore Hardware State** The third step in a context switch involves the restoration of the stalled context onto the ACS platform. Restoring the hardware state of a programmable logic circuit is the most difficult task of a hardware context-switch. Currently, few programmable logic vendors provide the capability of setting the state of internal registers and memories. Details on one implementation of this step will be described in the next section.

4. **Restart Hardware** The final step in a hardware context-switch is restarting the hardware. The hardware is restarted by enabling the global synchronous clock. As discussed earlier, configurable systems with clock control allow the clock to be stopped, restarted, and cycled as necessary. If the state is properly restored before the clock is cycled, the hardware process will restart and operate as though the circuit had never been removed from the hardware resources.

3.8.2 Writeback

As suggested earlier, it is necessary to restore the state of the programmable logic in order to perform a hardware context switch. However, programmable logic vendors do not provide the capability of setting the state of internal registers and memories. If state restoration is required within a programmable logic circuit, the user typically must add this capability into their circuit design.

One method for supporting this approach is instrumenting a scan chain into the user design. This approach allows a user to obtain and restore internal state by shifting the state bits in and out of the

programmable scan chain. While this approach provides state capture and restoration, it consumes large amounts of programmable logic resources. A study using scan-chain in Xilinx FPGAs[19] showed the area overhead to be from 60 percent to over 80 percent. Additionally, the large overhead may affect the overall timing of a user design.

A novel approach for setting the state of internal registers is to *modify* the original design to reflect the saved state of the hardware process. This modified design is then configured onto the resource to restore both the hardware configuration and hardware state onto the configurable resources. State restoration of flip-flops is implemented by changing the reset/preset behavior of each flip-flop in the design. Most FPGA vendors provide flip-flop libraries that support both preset and reset options. The state of internal registers can be set by setting the preset or reset mode of each flip-flop. For example, flip-flops with an initial "1" state are replaced by flip-flops in preset mode and flip-flops in the "0" state are replaced by flip-flops in reset mode. Once the design has been modified, a global reset is performed on the device and each register is restored to the appropriate state value.

Figure 12 demonstrates this writeback process for a four-bit reset register used in a design. When the circuit is first loaded, the global reset line is strobed and the four bit register is initialized to 0000. Later, when the design is stopped, it holds the value 0110. This value is saved as part of the state of the design, and the circuit is modified to reflect the state 0110. The two middle flip-flops are each changed from a flip-flop in the reset mode to a flip-flop in the preset mode. Later, when the design is restored and reset, the register will contain the value of its last known state. This techinque for presetting the register state is accomplished by changing the bitstream of the user design to reflect the changes in the circuit.

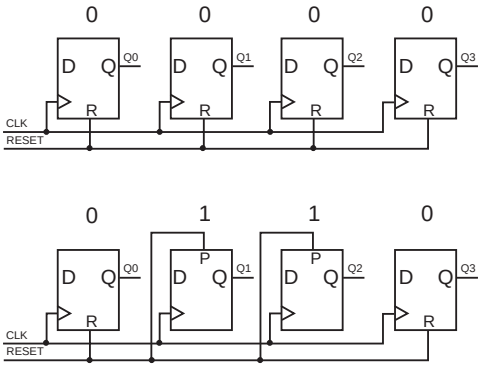


Figure 12: State Restoration on User Flip-Flops

3.8.3 Multitasking on SLAAC1-V

Multitasking was fully implemented and demonstrated with Xilinx Virtex FPGAs on the SLAAC1-V ACS platform [12]. The multitasking environment was developed to demonstrate the ability to support multiple users at the same time on the same hardware platform. This implementation of multitasking is based on a client-server architecture as shown in Figure 13. The host computer that contains the SLAAC1-V board provides a virtual connection to the SLAAC board through network sockets. This host "owns" the hardware and arbitrates the use of this hardware between competing hardware "clients". Hardware clients connect to the hardware server to request hardware-level service.

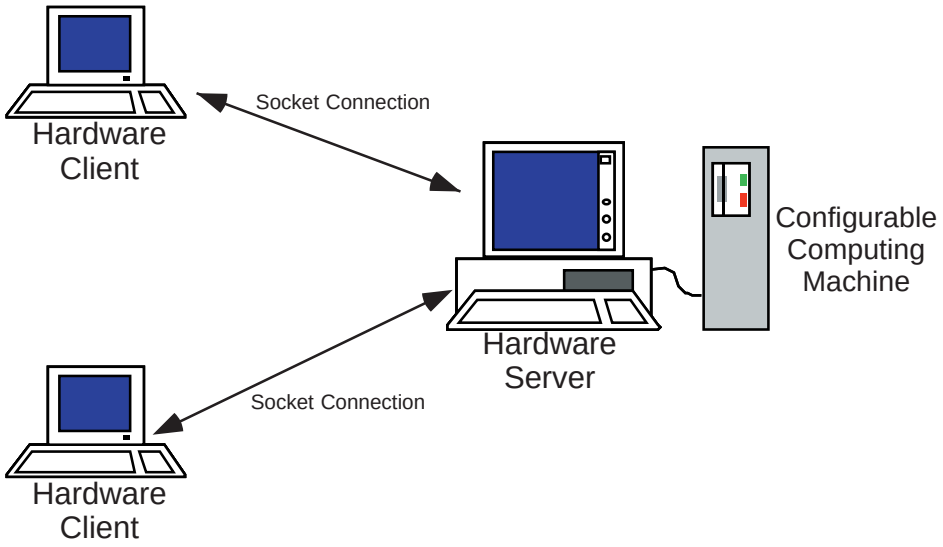


Figure 13: Client-Server Multitasking Architecture

Although hardware clients do not communicate directly with the hardware, they have access to the same API that is supported by the board driver. This API includes support for configuring devices, starting and stopping the system clock, reading and writing memory, save and store system state. The server receives requests to perform these operations from various users and executes these requests in order to maximize the availability and utilization of the hardware.

The performance of the SLAAC1-V multitasking environment was carefully measured to determine the speed at which a context-switch can occur. These results, shown in Table 2, indicate that a complete context switch consumes two and a half seconds. While this is significantly longer than the context switch of a modern processor, this context switch requires a tremendous amount of data transfer and storage. The operation consuming the greatest amount of time is the extraction of the memory state from the board. As the support for readback and writeback improve and the bandwidth of ACS systems increases, the time to perform a context switch will significantly decrease.

Table 2: Average Execution Time For Stages of a Hardware context-switch

Stage	**Exec. Time**	**% of Total**
Raw Readback	407ms	16.24%
State Extraction	423ms	16.88%
Memory Save	995ms	39.70%
Memory Restore	316ms	12.61%
Reconfiguration	365ms	14.57%
Total	2506ms	100%

Although multitasking was completed for the SLAAC1-V as a proof-of-concept demonstration, there is no API to provide support for multitasking for other hardware systems. The functions performed by a multitasking hardware system are very specific to the individual ACS platform and cannot be easily ported to other platforms. The memory architecture, host interface, and organization of programmable logic resources are unique for each platform and require custom functionality

for storing and restoring state. For some hardware platforms, multitasking cannot be supported due to the lack of appropriate I/O interfaces and readback support. While no general multitasking API is provided, the API shown in Program 3.5 to provide support for basic programmable logic checkpointing.

3.9 Remote Access

Originally the plan for remote access was to allow the user access through a web page or similar mechanism. We later discovered a better approach which uses capabilities built into Java to provide remote access to the boards directly in JHDL. This mechanism was built into the platform control facilities and will be discussed in Section 4.3.1.

3.10 Platform Control

At the bottom of Figure 1 is the "Platform Control Interface". This is the layer of software which the DSM uses to communication with hardware platforms. To streamline the creation of the DSM, this was written in a layered fashion. At the top (closest to the DSM) is a platform-independent layer which defines an API of generic routines for the DSM to control hardware. This layer contains routines such as ***configure()***, ***stepClock()***, and so forth. For each platform supported, a layer of platform-specific routines is then created which translate these API calls into the specific subroutine calls required to manipulate that platform. Due to the importance and complexity of the Platform Control Interface, a complete section on its structure and operation is included in Section 4 of this report.

3.11 Summary

Figure 1 provided a high-level view of the DSM structure. In this section we have reviewed the various parts of the DSM. Due to the importance of a few of the features of the DSM, they are described in their own sections which follow.

4 The Platform Control Interface

Because of the large degree of variability in the feature set of configurable computing platforms, the platform control interface is broken into two pieces: The first is a platform independent control interface and the second is a platform specific interface, which can be different from platform to platform. Using these two APIs allows for common features to be implemented in a platform independent manner, without precluding the use of platform specific features. When creating the platform APIs, there were four main goals:

1. High degree of flexibility. This includes flexibility in the features provided to the user, as well as flexibility in the views of the executing platform provided to the user.
2. New platforms should be able to be supported without adding extra features to the DSM.
3. When adding support for a new board model, common features should not have to be wholesale reimplemented.
4. In the case of platforms which provide only a C/C++ interface (which is the most common interface), the board model designer should not have to write JNI (Java Native Interface) code to provide support for the platform.

Given these goals, the common features needed to provide control and observability of FPGAs in common platforms have been provided as part of the platform independent APIs. Other features are provided as part of the platform specific APIs. The platform specific APIs are part of the *board model* provided for a target platform. Figure 14 provides a graphical representation of the relationship between various parts of the platform control interface.

The figure shows the main parts of the platform interface. Two of these parts, the Browser and the DSM facilities were discussed previously. In addition to these, JHDL provides two other APIs to help support controllability and observability of the platform. These are the Platform Independent Interface (PII) and the Hardware Control Interface (HCI). There are also five main pieces of the board model that must be written in order for JHDL to support a new platform. These parts are shown in gray in Figure 14. Each of these seven parts are discussed below.

4.1 The Platform Independent Interface (PII)

As reported in Section 3.5, the simulator in the DSM allows execution data from the FPGA to be back annotated into the simulation data structures. In addition to this, the platform interface allows the simulator to have some control over the circuit clock. To enable these features, the DSM provides two interfaces that a board model must implement in order for the simulator to be able to provide this functionality. The interfaces are the HardwareInterface and the NativeReadBackInterface. These interfaces are shown in Program 4.1 and 4.2.

The HardwareInterface provides methods to advance the hardware clock and to get the current values of the state elements in the executing circuit. This is done by passing the board model a list of all elements in the circuit that can be externally updated. The board model then uses the ReadBackManager as discussed in Section 3.6 to extract the state of each element from the readback bitstream. The NativeReadBackInterface is used to provide the ReadBackManager with

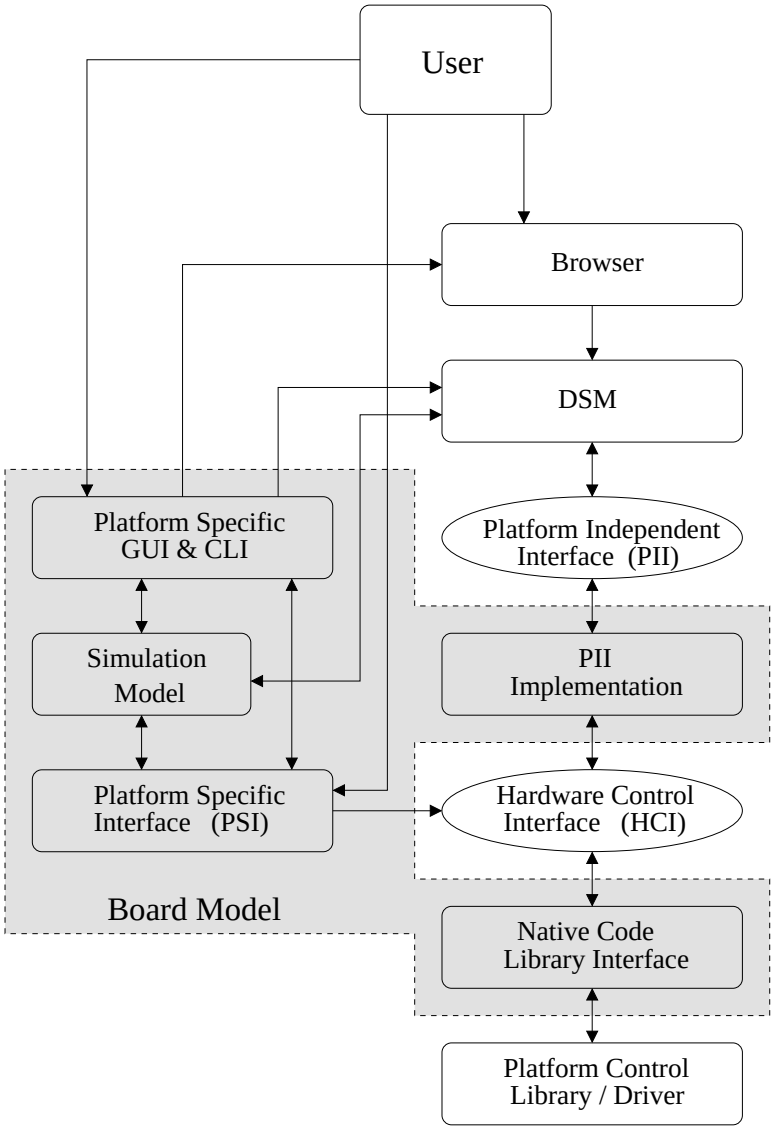


Figure 14: API Layers Used in JHDL For Platform Control

the readback data stream from which the circuit state is extracted. Together, these two interfaces provide the ability to both control and observe the state of circuits executing on an FPGA. When creating a new board model, these methods must be implemented in order to enable hardware mode for the new platform.

4.2 Implementation of the Platform Independent Interface (PII)

When writing a board model, the author must implement the interfaces described in the previous section. The purpose of the implementation is to translate the platform independent calls into the platform specific implementations, targeting the HardwareControlInterface, discussed next.

Program 4.1 The HardwareInterface API

```
/*
  Allows the simulator to advance the hardware clock.
*/
void stepHardwareClock( int cycles );

/*
  Provides the simulator with the state of the requested
  circuit elements.
*/
StateObject getHardwareState( ExternallyUpdateable[] eCells,
                              LargeExternallyUpdateable[] leCells,
                              Checkpointable[] cCells );
```

Program 4.2 The NativeReadBackInterface API

```
/*
  Request the board model to readback the state of the
  specified FPGA.
*/
int nativeReadBackPE(int peNum, byte [] bitstream);

/*
  Finds the specified JHDL cell by name.  This method
  allows the board model to modify the name if necessary
  before finding the cell.
*/
Cell findJhdlCell(String cellName);
```

4.3 The Hardware Control Interface (HCI)

The DSM provides a generic interface, through which platform control calls can be mapped. This is call the HardwareControlInterface (HCI). The HCI is a flexible interface which serves to provide a consistent interface to hardware for different platforms. The HardwareControlInterface is used by both the PII and the PSI to access the hardware. This interface was created for two reasons:

1. This interface allows the DSM to provide a pre-built JNI library which maps the Java API to a C++ API. This makes it possible to create a new board model which uses a C/C++ interface library, without having to write JNI code (which can be quite tricky to get right). Figure 15 shows a graphical representation of this. The JNI library includes the code to map Java data types to C++ data types and simply acts as a translation layer between the two APIs.
2. This common interface allows the DSM to provide a clean, easy to use mechanism to provide remote access to the board. This is discussed in Section 4.3.1.

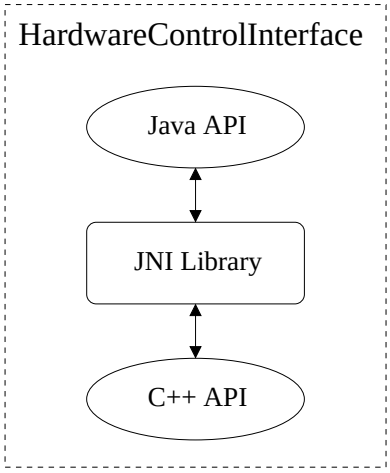


Figure 15: Structure of the HardwareControlInterface

The Java and C++ sides of the HardwareControlInterface are shown in Programs 4.3 and 4.4, respectively. The implementations of the Java methods are provided in the JNI Library (Note that all methods in this class are labeled **native**. This means that the methods are implemented using JNI.). The board model writer is responsible for implementing the methods in the C++ class, which will interface to the platform control library. Note that the two interfaces are identical, with the exception of some translations which must be done because of slight differences in the languages. This translation is handled in the JNI library provided with the DSM.

4.3.1 Remote Access

Remote access is provided by using the RMI (Remote Method Invocation) feature of Java. To enable remote access, the DSM uses an RMI aware version of the HardwareControlInterface, called NetworkHardwareControl. The interface provided by both of these are identical. Because the DSM provides a generic API which can be used to implement the platform specific features of a platform,

Program 4.3 The Java HardwareControlInterface API

```
public class HardwareControlInterface {
  // Board methods
  public native int open();
  public native int close();

  // Clock methods
  public native int stepClock(int clock, int steps);
  public native int setClockFrequency(int clock, float freq);
  public native int freeRunClock(int clock);
  public native int stopClock(int clock);

  // FPGA program/readback methods
  public native int program(int fpga, byte[] data);
  public native int readback(int fpga, byte[] data);
  public native int writeback(int fpga, byte[] data);

  // Memory methods
  public native int getMemoryWidth(int set, int memory);
  public native int setMemory(int set, int memory, int address, int length, byte[][] data);
  public native int getMemory(int set, int memory, int address, int length, byte[][] data);

  // Register space methods
  public native int getRegisterWidth(int set, int register);
  public native int setRegister(int set, int register, byte[] data);
  public native int getRegister(int set, int register, byte[] data);
}
```

Program 4.4 The C++ HardwareControlInterface API

```
class HardwareControlInterface {
public:
  // Board methods
  virtual int open() = 0;
  virtual int close() = 0;

  // Clock methods
  virtual int stepClock(int clock, int steps) = 0;
  virtual int setClockFrequency(int clock, float freq) = 0;
  virtual int freeRunClock(int clock) = 0;
  virtual int stopClock(int clock) = 0;

  // FPGA program/readback methods
  virtual int program(int fpga, signed char *data, int data_len) = 0;
  virtual int readback(int fpga, signed char *data, int data_len) = 0;
  virtual int writeback(int fpga, signed char *data, int data_len) = 0;

  // Memory methods
  virtual int getMemoryWidth(int set, int memory) = 0;
  virtual int setMemory(int set, int memory, int address, int length, signed char **data, int data_len) = 0;
  virtual int getMemory(int set, int memory, int address, int length, signed char **data, int data_len) = 0;

  // Register space methods
  virtual int getRegisterWidth(int set, int register) = 0;
  virtual int setRegister(int set, int register, signed char *data, int data_len) = 0;
  virtual int getRegister(int set, int register, signed char *data, int data_len) = 0;
};
```

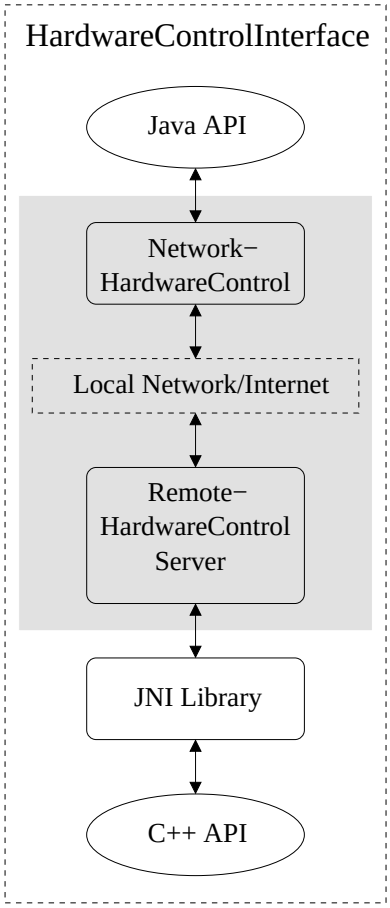


Figure 16: API Layers Used in JHDL For Remote Platform Control

remote access can also be provided in a generic manner, using this interface. This functionality is built into the NetworkHardwareControl.

As shown in Figure 16, on the other end is a server which takes the RMI calls and translates them back to C++ calls. This C++ code is identical to that used for local access. The server, which is written in Java to take advantage of RMI, can be simple or sophisticated. We demonstrated a simple server for the SLAAC1-V board, which allowed only one person access to the board at a time. The server could be more complex and use the multi-tasking work (see Section 3.8) to allow multiple users access at the same time in a time-sliced manner.

4.4 The Platform Specific Interface (PSI)

The platform specific interface allows for a high degree of flexibility in supporting new platforms. The author of a board model is free to add any appropriate feature to the platform specific interface. The feature set in the PSI includes any feature of the target platform which is not provided by the PII, but which the board model writer would like to provide to the end user. This typically includes the ability to read and write external memories, FIFOs and registers provided as part of the platform, but may include any other feature of interest.

4.5 Platform Specific GUI and CLI

In addition to using the features built into the default viewers, the board model can also provide platform specific graphical interfaces and textual commands. The added viewers provide the user with a faster view on what is happening in the hardware. This is done in a way which make sense for the target platform. A typical viewer brings up a graphical view of the board which is similar to that provided by a block-diagram of the board. Shown in the view are the processing elements (PE's) on the board as well as the memories. Double-clicking a PE brings up a dialog which allows the PE to be configured with a bitstream. Double-clicking on a memory brings up a dialog which allows the memory contents to be loaded from a file. Shift-clicking on a PE brings up a schematic view of the contents of the design currently configured onto that FPGA; shift-clicking on a memory brings up a memory contents browser window to enable examining the contents of the memory on the board. Other controls allow enabling or disabling readback for a given PE, resetting a given PE, setting or viewing the contents of FIFO's, and viewing the state of the LED's attached to the board.

4.6 Simulation Model

In addition to the interfaces created for the board model, the board model must also provide a simulation model for the platform. This includes modeling board level elements as well as circuit elements included in the FPGA. These can include, but are not limited to models for external RAMs, FIFOs, board registers, etc. The simulation model provides an environment for the designed circuits to execute in to verify the correct function of the circuit. This model also provides the basis for the netlist environment, which creates a netlist that can be run through the vendor tools to create the programming bitstream for the FPGA.

4.7 Native Code Library Interface

The native code library interface translates the API calls for the C++ side of the Hardware Control Interface to a platform specific implementation which makes calls into the interface library provided by the platform vendor.

5 Debug Circuitry Synthesis

An example of the use of the DSM is in instrumenting existing designs for debug. One approach to this is to *temporarily* instrument the design with special debugging hardware to provide the designer with the design observability and controllability desired. Among its many applications, instrumenting FPGA designs for debugging can provide observability into design execution while the design executes at or close to normal speed or it can even provide similar observability to that provided by readback. The reprogrammability of SRAM-based FPGAs makes this option reasonable since, once the validation and debugging processes have ended, the debugging hardware can be removed from the final design, eliminating the temporary area and speed costs of embedding debugging hardware.

Instrumenting designs for debugging has generally been performed before the design's physical implementation, i.e., using the design's source description (HDL or schematic) or its netlist. Unfortunately, this means that each debugging modification requires almost a complete recompilation of the design so that it can be executed in hardware—a process which takes tens of minutes to hours to complete for significant designs. With this large time cost, a designer may only be willing to instrument designs for debugging under extreme circumstances or the designer may instrument the design with several instances of debugging hardware all at once so the reimplementation costs are only paid once or a few times. Of course, in the latter case, all of the debugging hardware is not being used for each execution of the hardware, meaning that this case requires a greater area cost than if the instrumentation was done as needed for each debugging task.

In this section we describe our approach for directly modifying bitstreams for debugging (after implementation). To provide a concrete description of the process, we will use the instrumentation of FPGA designs with embedded logic analyzers (ELA's) as a detailed example. We will then describe several other ways we have employed bitstream instrumentation for adding other debugging hardware into user designs. These examples will be less detailed since they share much in common with the ELA example.

Commercial packages exist which provide logic analyzer cores which can be embedded into FPGA designs as well as software for controlling these ELAs. However, as a demonstration of bitstream instrumentation for debugging, we designed our own simplified logic analyzer cores which can be modified easily at the bitstream level and we added experimental support for these ELAs into the JHDL environment for Virtex FPGAs. When starting a design execution, the user can select the wires to be traced and those used for triggering and, within seconds to a few minutes, connect his or her circuit up to embedded logic analyzers for debugging purposes. We will first describe our ELA core and then describe the implementation of our bitstream instrumentation scheme which uses JHDL, JBits, and JRoute. JBits and JRoute are tools available from Xilinx Corporation. They provide the ability to construct a bitstream for a Xilinx FPGA without going through the conventional place-and-route CAD tool cycle. They do this by providing an API of function calls which can be used to set specific LUT contents, turn on or off routing switches, etc. In addition to supporting the creation of bitstreams directly, they also support the modification of existing bitstreams. In the work described in this section, they are used to modify the bitstream associated with an existing design by modifying LUT contents (JBits) and wiring LUT's and FF's together (JRoute). This can all be done in a matter of a few seconds and thus is suitable for use in an interactive debug environment. JBits and JRoute are both written in Java and so were easy to interface with the JHDL environment.

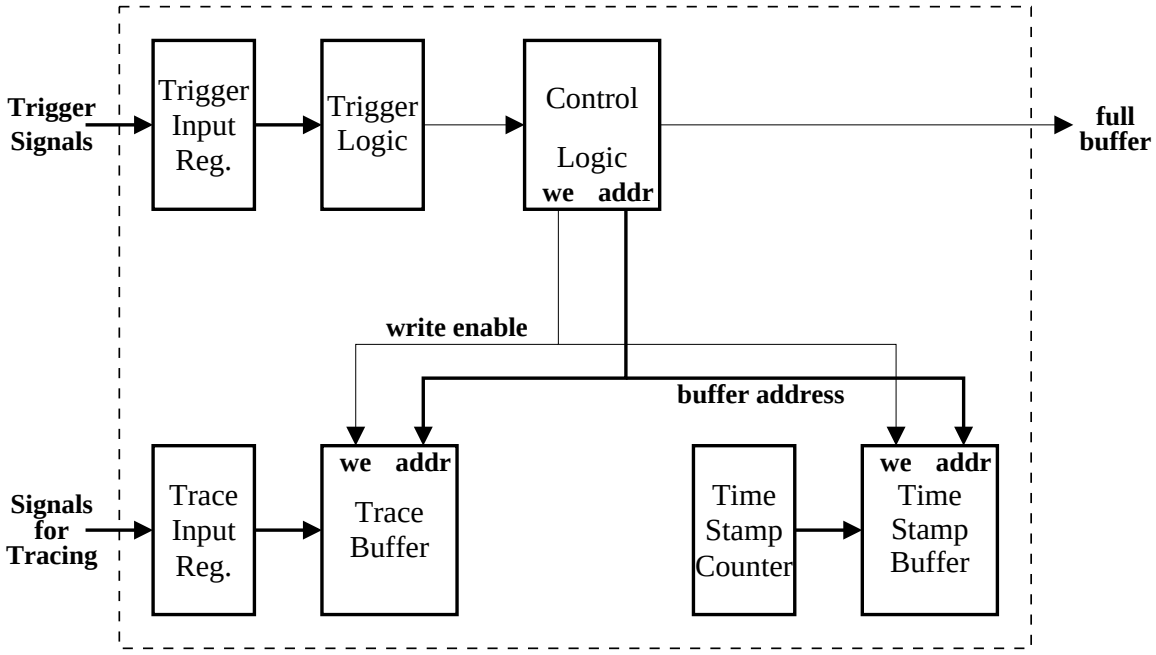


Figure 17: Bitstream-Modifiable Logic Analyzer Core

5.1 A Bitstream-Modifiable Logic Analyzer

A simplified block diagram of one of our logic analyzer cores is provided in Figure 17. Not shown in the diagram is the buffer dumping logic which allows several logic analyzers to be cascaded so that a single debugging port can be used to collect the data from multiple logic analyzers. This particular logic analyzer core is for sparse signal sampling, i.e., whenever the trigger condition is true, the ELA records the trace signals' values along with a time stamp representing when the event occurred—the reason for having both trace and time stamp buffers in the design.

Distinguishing this logic analyzer from its commercial counterparts, we designed it for bitstream instrumentation in two ways. First, the signals used for triggering and the triggering function can be defined by modifying only the design's bitstream. This means that the trigger condition logic can be programmed and the signals used for generating the trigger function can be connected to the logic analyzer just before circuit execution. Figure 18 shows the simple combinational trigger logic we used in our demonstration system. Though this combination of 5 4-input LUTs cannot perform all functions of 16 inputs, it can perform many useful logic functions and can be easily programmed using JBits. Second, the signals to be traced can also be connected to the state analyzer via bitstream modification just prior to execution using JBits and JRoute.

In our demonstration system, the logic analyzers are actually inserted prior to netlisting but are not connected to the user's circuitry in the netlist. At the time we created our demonstration system, JBits and JRoute did not fully support BlockRAMs or IOBs so we had no other choice. In our system the only bitstream-modifiable features of the logic analyzers are their connections to user signals and the trigger logic functionality. With the latest versions of JBits and JRoute (JBits distribution versions 2.5 and later), adding the logic analyzers directly into the design's bitstream should now be possible, providing the ability to configure buffer depths *after* place and route have executed as well as allowing the trigger logic to be more general and flexible. This is an obvious next step for our system.

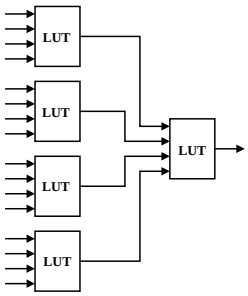


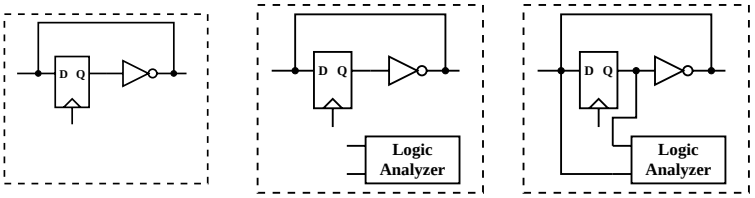
Figure 18: Bitstream Configurable Trigger Logic

Note that the Xilinx tools would normally optimize away logic not connected to anything in a user design (such as our logic analyzers). To prevent this a number of things were required. First, the inputs to the logic analyzers (trigger and data signals) were registered and logic placed in front of the registers to prevent their removal by the logic trimming tool. Second, the trace buffer outputs were tagged with *save* attributes in the netlist to prevent them from being optimized away. As side benefits, the inclusion of input registers also simplified the use of JBits and JRoute to connect user signals to the logic analyzer inputs and reduced potential critical path delays in the system.

5.2 The Demonstration System

As discussed above, the system we created to test bitstream instrumentation for debugging was based on three main software technologies: JHDL, JBits, and JRoute.

Process Overview One or more logic analyzers are automatically inserted into the user's design prior to netlisting. Figures 19(a) and (b) represent the circuit before and after this is done. The logic analyzers are fully visible to the user and fully simulate, but do not connect to anything at this point. The user then generates an EDIF netlist which is run through the Xilinx FPGA implementation tools.



(a) Original Design

(b) Before Netlisting

(c) After Instrumenting Bitstream

Figure 19: Instrumenting a User Circuit With a Logic Analyzer

After this and before execution, we extract several things from the Xilinx MAP report file (.mrp) and the XDL representation of the physical design, including:

1. the physical FPGA routing resources used for each net,

2. the physical locations of all flip-flops,
3. the locations of the sources of all physical nets,
4. and the physical locations of the trigger logic LUTs.

Once the bitstream has been generated and the user is ready to execute the design using JHDL's hardware mode, the design is loaded into the JHDL system and the bitstream is downloaded into the FPGA. If, at this time, the designer decides to use the logic analyzers, he configures the wires for the logic analyzers to trace by selecting the signals in a schematic window and executing the "Hardware Trace Wire(s)" command. This is shown in Figure 20. Next, the user brings up a dialog box which allows him to select additional wires for tracing and specify the trigger conditions desired. This is shown in Figure 21.

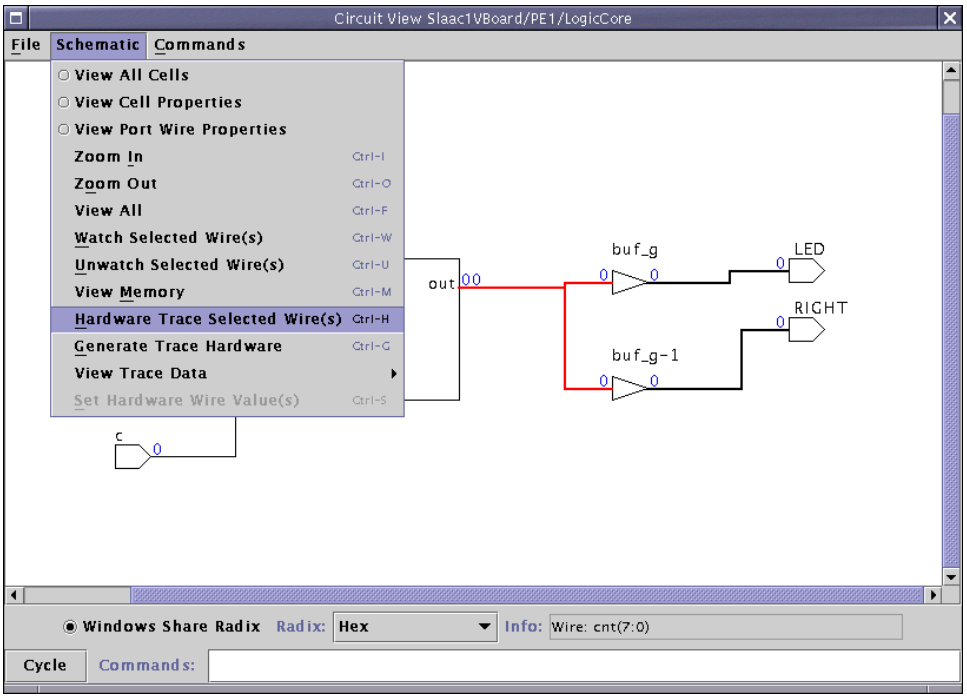


Figure 20: Selecting a Wire to Trace With ELA

When satisfied with the configuration of the logic analyzers, the user commits this configuration to the design. This does three things: (1) the internal JHDL representation, or logical database, is automatically modified to reflect the logic analyzer's new configurations (the user can now see the logic analyzers and their connections to his design in the schematic windows), (2) the configuration bitstream for the user's design is automatically modified using JBits and JRoute to implement the new logic analyzer configurations, and (3) the new configuration bitstream is downloaded into the FPGA. Finally, the user executes the modified design in hardware and examines the collected traces. This is done using specialized windows in the Unified Browser and is shown in Figure 22. Note in the Figure that multiple signals ($cnt[7:0]$ and $RESET$)were collected for each time stamp during which the trigger condition was true.

A full discussion of these and other debug synthesis topics and the resulting performance achieve are provided in the PhD dissertation of Paul Graham [4] and in the paper [2]. It should also be noted

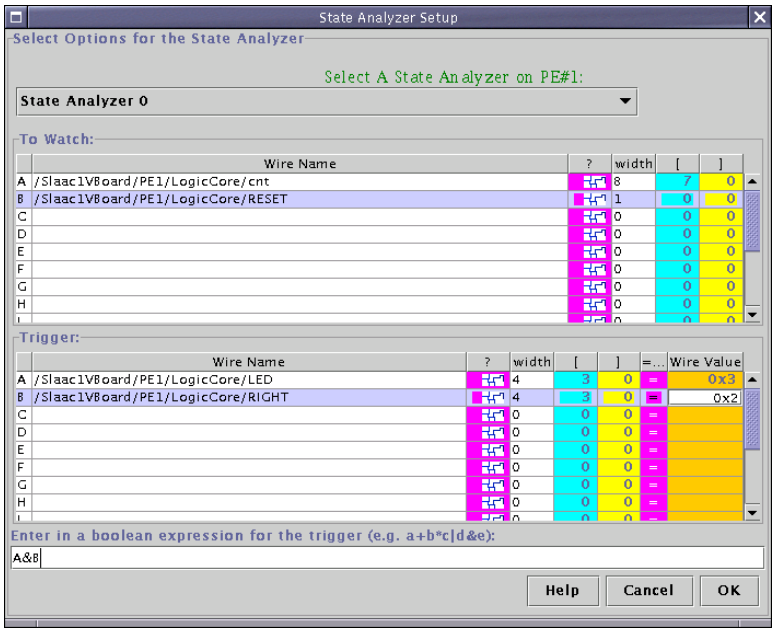


Figure 21: Selecting Additional Wires and Specifying ELA Trigger Condition

that the work described in this section and in the reference papers all took the form of proof-of-concept demonstrations. As such, no publicly accessible API's in the JHDL system exist and thus none have been documented in this report.

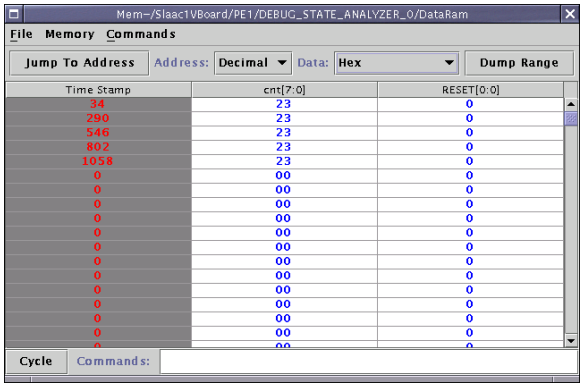


Figure 22: Collected ELA Results

6 The Data Interchange Specification, Tools, and API

As mentioned previously, the debug environment developed as a part of this research was built on top of the JHDL design environment. The bulk of the use of the debug environment conducted at BYU was done using circuits originally created in JHDL. However, it was determined at the outset that the debug environment was to be usable on non-JHDL designs. To that end, a data interchange mechanism was developed so that such *external* designs could be imported into the debug environment and the power of the DSM and Unified Browser applied to the debug of those designs.

The decision was made to use EDIF for the data interchange format. All major HDL synthesis and schematic editing tools have the ability to generate EDIF. All major FPGA vendor tools accept EDIF which they then map, place, and route onto the physical FPGA device. The first step in the research was to write an EDIF parser so that EDIF files could be parsed into the DSM's internal circuit representation. Once the internal circuit representation was created, the design could be simulated and executed just like any JHDL-created design.

6.1 Importing EDIF Designs into the Debug System Using the EdifParser

The EdifParser makes it possible for you to use an already existing EDIF file as a JHDL module. It will do the following:

1. Take the EDIF file and read it.
2. Identify all the cells and elements in the EDIF file and collect information about those cells and elements.
3. Translate those cells and elements into JHDL primitives.
4. Build the actual circuit data structures so you can look at it using Unified Browser, simulate it, netlist it, etc.

Importing an external design is done by creating a JHDL design. However, instead of instancing primitive cells in the normal JHDL way to build a circuit, the circuit is built by opening and parsing an EDIF file.

Consider the code shown in Program 6.1. This is conventional JHDL code with a few additions. First, the EDIF parser code is imported at line 5. Second, the port interface for the external EDIF is defined starting at line 19. Finally, the EDIF parser is invoked on file *"add.edn"* to import the EDIF (line 25). Once this code executes, the contents of the EDIF will be represented in the DSM's internal circuit representation data structures. The circuit can then be instanced as part of a larger design, netlisted, simulated, or executed in hardware.

6.2 EDIF Coverage and Compatibility Issues

EDIF is a very large language, containing many features not generally used for simply describing logic netlists. To make the parser of manageable size, the EDIF parser was written to recognize the entire language as input but to only accept a subset of that for building circuits. If the EDIF

Program 6.1 Importing an External Design

```
( 1) import byucc.jhdl.base.*;
( 2) import byucc.jhdl.Logic.*;
( 3) import byucc.jhdl.Xilinx.*;
( 4) import byucc.jhdl.Xilinx.Virtex.*;
( 5) import byucc.jhdl.parsers.edif.sablecc.*;
( 6)
( 7) public class add_Import extends Logic {
( 8)
( 9)   public static CellInterface[] cell_interface = {
(10)     in( "a", 4), in( "b", 4), out( "q", 4),
(11)   };
(12)
(13)   public add_Import(Node parent, Wire a, Wire b, Wire q) {
(14)     super(parent);
(15)     connect( "a", a);
(16)     connect( "b", b);
(17)     connect( "q", q);
(18)
(19)     EdifPortInterface[] portWires = {
(20)       EdifPortInterface.addPort("a", a),
(21)       EdifPortInterface.addPort("b", b),
(22)       EdifPortInterface.addPort("q", q),
(23)     };
(24)
(25)     EdifParser.parse(this, "add.edn", portWires, "Virtex");
(26)
(27)   } // end constructor
(28) } // end class
```

file contains language features which are not part of that subset, the parser will provide a message that it is ignoring those features. However, the generation of such warning messages are essentially non-existent since the parser has been augmented over time to accept the EDIF subset used by all major CAD tools. The EDIF parser has successfully been used on EDIF from a variety of sources: SYNOPSYS VHDL, Synplicity VHDL, Viewlogic Schematics, JHDL-generated EDIF, and the Xilinx Coregen tool.

6.3 On-Line Documentation on Data Interchange

While the above has provided an overview of the EDIF import capability, the JHDL on-line user's manual contains a full chapter on importing external designs into the debug system. This can be found on the WWW at:

http://www.jhdl.org/docs/docs/usersManual/exportImport.html

7 Using High-Level Tools with the DSM

7.1 Source Level Debugger

The source level debugger [5] is an example of a high-level tool which uses the services provided by the DSM. It uses the DSM services to provide software-like debugging techniques for the debug of circuits synthesized from high level source (Java code for example). An overview of how the various tools used for source level debugging are related to one another is shown in Figure 23. The goal of the source level debugger is to extend the capabilities of JHDL, allowing the debugging of synthesized circuits. This means that the debugging of the synthesized circuit (at the logic level) is done in the context of the original high level source code. This section will summarize the feature set of the source level debugger and how the DSM services are used to help provide them. Details of this work can be found in [8].

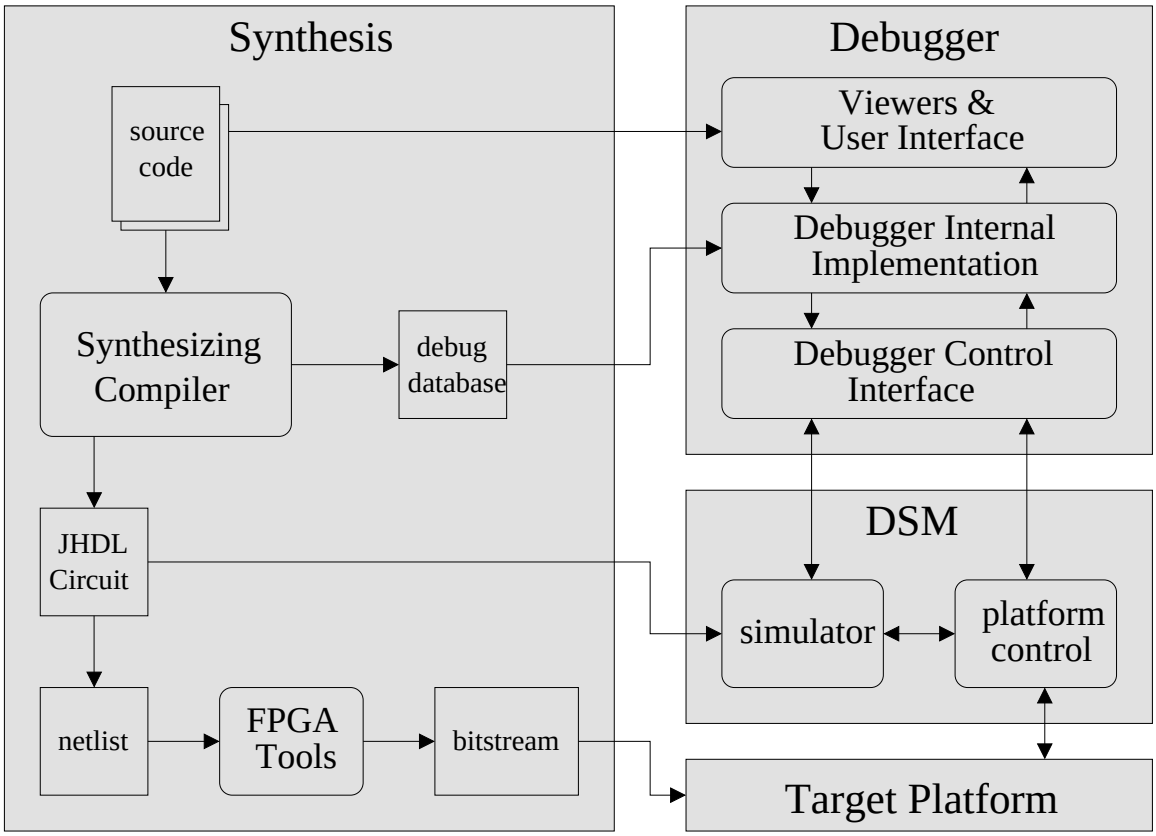


Figure 23: Overview of Source Level Debugger

The source level debugger was designed to be used in conjunction with a synthesizing compiler such as the Sea Cucumber compiler[18, 7]. In order to enable source level debugging, the synthesizing compiler must be modified to create a debug database along with the synthesized circuit. The debug database records information about how the high level source code was mapped to the final, logical circuit. The final mappings are built up from incremental mappings made at three levels of abstraction, shown in Figure 24: the source level, the CFG/DFG (control flow graph/data flow graph) level, and the hardware level.

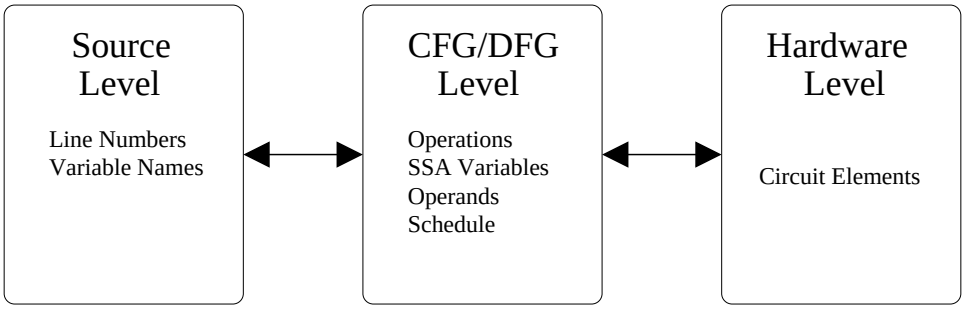


Figure 24: Levels of Abstraction Used in the Debug Database

The source level of abstraction is the level at which the debugger communicates information to and receives control instructions from the user. It consists of line numbers and variable names. The CFG/DFG level of abstraction is representative of the data structures used by the compiler and contains information about operations, static single assignment (SSA) variables (operands), and schedules. The hardware level consists of circuit states and circuit elements such as registers, memories, wires, etc. This is the level at which the system observes and controls the running circuit. In addition to the mappings between these levels of abstraction, there are also mappings found withing the CFG/DFG level of abstraction. The incremental mappings can be reused by chaining them together in different ways to produce the final mappings required by the debugger.

These mappings must account for changes in both the control- and data-flow of the program made by optimizations. The primary optimizations looked at for this work were: static single assignment, predication, block merging and instruction scheduling. The modified compiler now generates the synthesized circuit and the debug database.

The information in the debug database is used to implement the feature set of the debugger which includes single stepping, breakpointing, location of current execution points and the watching and setting of variable values. In addition to these general features, the debugger provides two modes of operation. Each of these modes takes a different approach to the definition of a single step. The first takes a hardware-oriented approach to the definition of a single step and the second takes a software-oriented approach. The first mode is called clock step mode and defines a single step as advancing the circuit on clock cycle. The second, source step mode defines a single step as advancing execution one line in the source code, just as in a software debugger.

In clock step mode, the debugger shows truthful behavior and does not attempt to hide the results of optimizations. Each single step advances the clock a single cycle. The debugger shows the user all lines in the source code which are currently executing in the circuit. It also allows the user to view and set the value of variables found in the source code.

In source step mode, the debugger shows expected behavior and uses Virtual Sequentialization to make the reordered, parallelized operations appear to execute sequentially, in the order expressed in the source code. This mode attempts to hide the details of the optimizations from the user and allows the user to concentrate on debugging the functionality of the application.

As a testbed, we used the Sea Cucumber (SC) synthesizing compiler targeting the SLAAC1-V configurable computing platform. To accomplish this, SC was modified to build the debug database as part of the compilation process The synthesizer was also modified to add additional debug circuitry to the design. Among this additional circuitry are breakpoint units, buffering units and replay units.

Each of these enhances the controllability and/or observability of the debugger. A screen shot of the SC Debugger is shown in Figure 25.

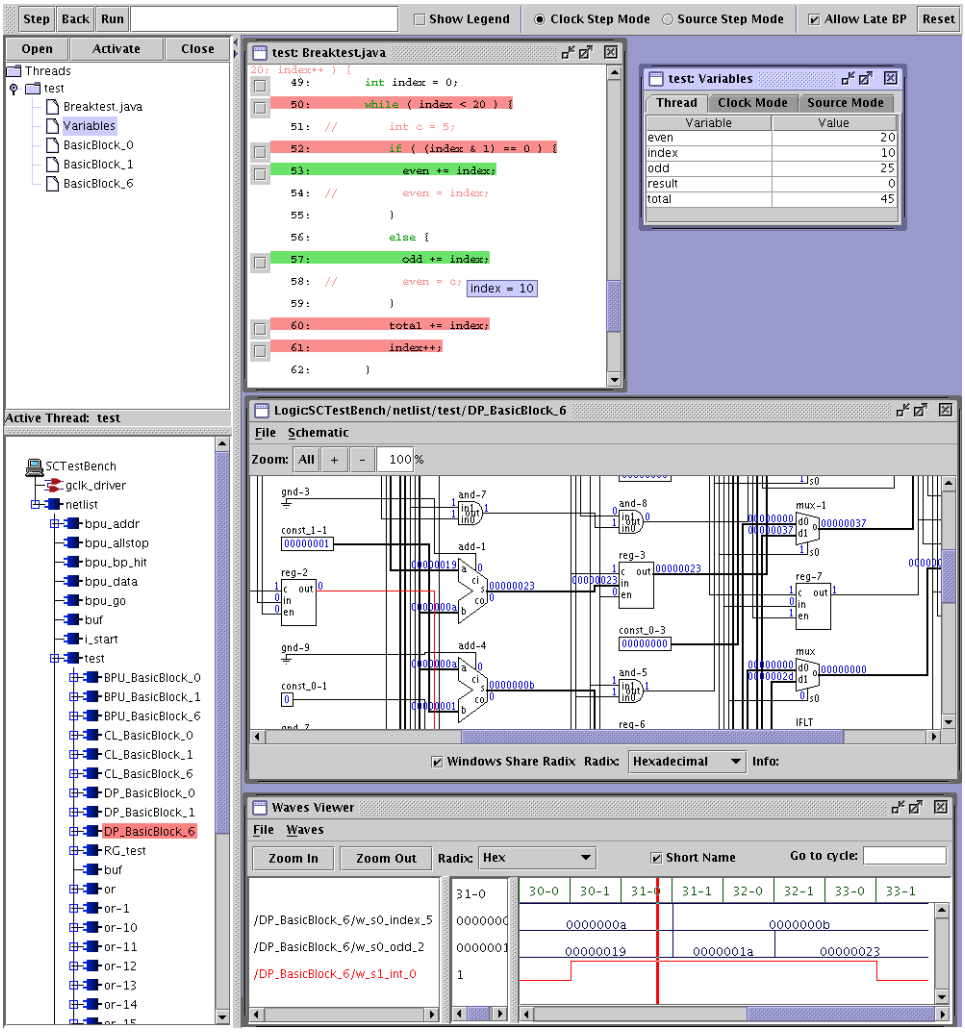


Figure 25: Screen Shot of the Source Level Debugger

To provide platform independence, the SC Debugger defines a high level API for interacting with the synthesized circuit (this is the Debugger Control Interface pictured in Figure 23). To support a new platform, a user must implement the methods in this API to provide platform specific controllability to the debugger. When implementing this API, the author uses the APIs available in the DSM and platform control interface to implement the methods to provide controllability and observability of the circuit. Hardware mode in the debugger is enabled by using hardware mode in the DSM. The debug database provides the mappings from source code to the logical circuit and the DSM provides the logical to physical mappings. Thus, a complete mapping from source code to the physical device is created. This allows the debugger to be used to debug the actual execution of a synthesized circuit in the context of the original source code.

The flexibility of the DSM was key in implementing the source level debugger. The difficult work of mapping logical circuit elements to physical location in the FPGA is done by tools provided with the DSM. The DSM also provides the runtime support for reading the state of the executing circuit

and annotating this information into the logical description. This allows the source level debugger to use a common interface for both simulation and hardware debug.

7.2 A Debug Environment for Hybrid FPGA/CPU Systems

Reference [14] (included in the Appendix) introduces another use of the debug system for use with a high level tool. From the abstract of that paper:

> The combining of one or more CPU's and an FPGA fabric on the same die is growing in popularity. Such programmable system-on-chip (PSOC) systems promise performance and development time advantages over conventional technology. They include embedded CPU's which can be characterized as either hard cores (designed into the PSOC at manufacture time) or as soft cores (designed by the end user of the FPGA using existing FPGA resources). Current debug approaches for PSOC's are derived from ASIC and embedded software debug methods and fail to take advantage of the special feature of PSOC's that can simplify debugging. In addition, ASIC approaches focus on the design at the gate level and provide little to help the designer identify errors in the embedded software parts of the system. Embedded software approaches focus on the use of symbolic debuggers to debug the embedded software but ignore the FPGA parts of the system. In a PSOC design, design errors may occur in many different places — the design of the CPU, the embedded software, the FPGA-based parts of the design, or the interfaces between these various parts. This paper presents a flexible tool that allows the user to dynamically adapt the CAD tool's behavior to the level of detail needed to track down design errors in PSOC designs. It provides for the creation of and coexistence of software source debuggers and gate level debug tools, all incorporated into the same debugging environment. A prototype PSOC debugging system based on a derivative of the JHDL CAD tool is presented which illustrates the range of debug support in both simulation and hardware execution modes that can be provided for PSOC debug. Extensions to the tool to support a wider range of embedded processors and GNU compiler tools are discussed along with conclusions and future work.

The remainder of the paper then describes use of the DSM and Unified Browser to aid in the debug of a system consisting of a soft-core CPU running in an FPGA fabric. Such a system has C-code running on the soft-core processor while the remainder of the FPGA fabric is used for gate-level logic which works in conjunction with the CPU to complete the computation.

To create such a hybrid debug environment on top of the DSM and Unified Browser a number of tasks were completed:

- A set of subroutines were created to parse the symbol table information produced by the assembler and compiler for the chosen CPU.
- A set of custom GUI windows were added to the Unified Browser to provide CPU-specific views of the CPU parts of the design while the original Unified Browser windows were retained to provide gate-specific views of the remainder of the design. The CPU-specific windows interacted with the DSM via the same API's that the Unified Browser uses.

This *extended JHDL* system was then demonstrated in the debug of a number of sample designs.

7.2.1 CPU-Specific Browser Windows

The first CPU-specific window added to the browser was a CPU control view window. As is common with embedded processor tools, this window shows the state of the CPU by providing the values of its registers, program counter, instruction register, etc. The creation of this window was a simple matter given the DSM API's which exist for providing circuit state information to browser windows. This window is shown in Figure 26.

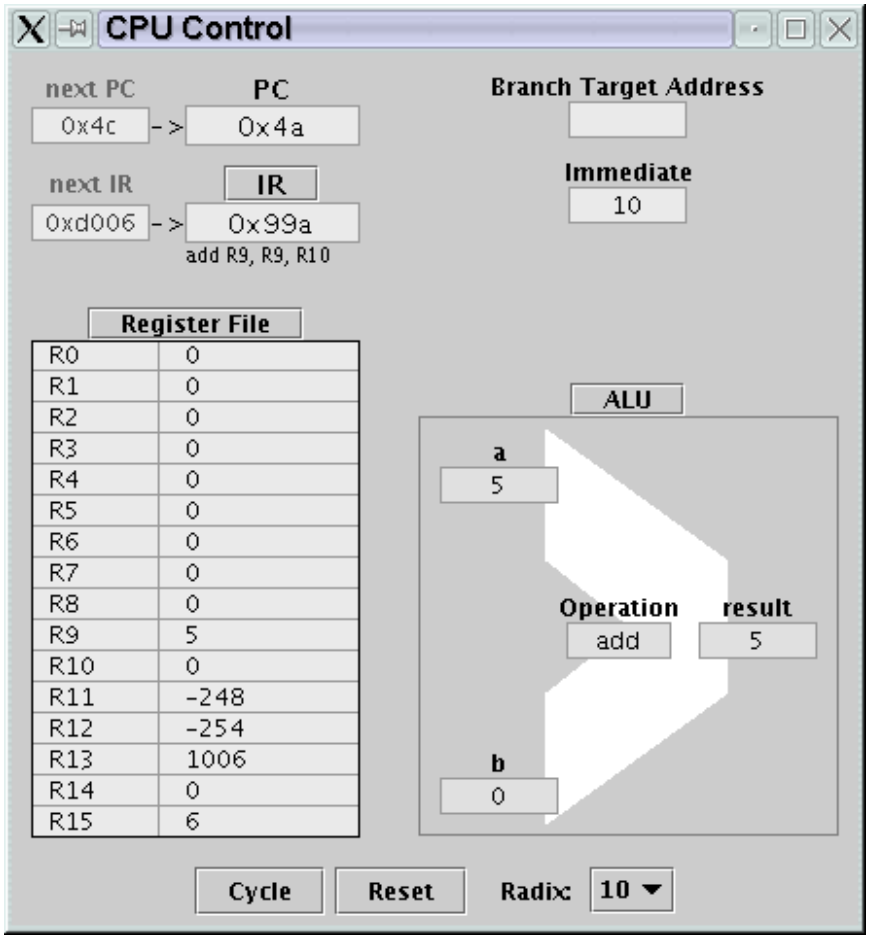


Figure 26: CPU Control Viewer Showing the CPU in the Middle of an Add Instruction

The second CPU-specific window created is an assembly source code browser. It uses the symbol table information created during the C-code compilation process and each clock cycle, correlates the hardware value for the PC (obtained through DSM API calls) with the original assembly source code. As a result the user can see, on a cycle by cycle basis, where in the assembly source code the soft-core CPU program is executing. In addition, he can set breakpoints based on assembly source code line number. This window is shown in Figure 27.

The final CPU-specific window created is a C source code browser. Like the assembly source code window, it uses symbol table information produced by the compiler to correlate the current value of the CPU's PC with lines in the source code. It also provides a breakpoint capability. This window is shown in Figure 28.

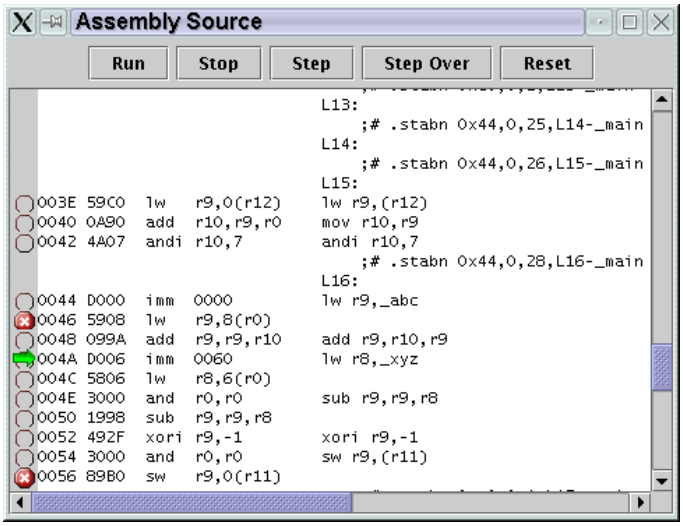


Figure 27: Assembly Source Code View Showing Breakpoints and Current Line of Execution

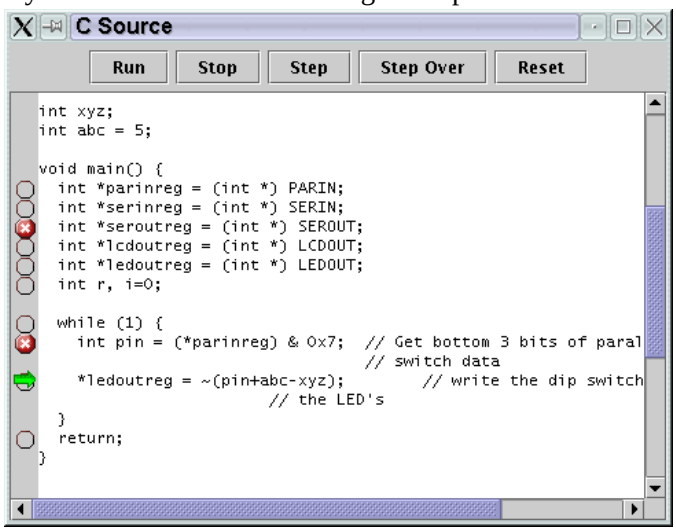


Figure 28: C Source Code View Showing Breakpoints and Current Line of Execution

The resulting browser consisting of original Unified Browser windows and new CPU-specific windows is shown in Figure 29.

7.2.2 Extensions to the Hybrid Debug Environment

A second version of the hybrid debug system described above was then created. The major difference with this second version was that it supported the use of the GNU *gcc* and *gdb* tools for compilation and debug (the original version used LCC instead). This required the creation of *gdb* stub routines so that *gdb* could be used for all source code interaction. The result was that all of the power of *gdb* could be brought to bear for the C source code debugging. This allowed not only the setting of breakpoints and the viewing of source code, but also allowed for the watching of C source code variables as the program executed (the *gdb* code base contains routines to enable the viewing of structs, arrays, etc). The resulting system is documented in detail in [15].

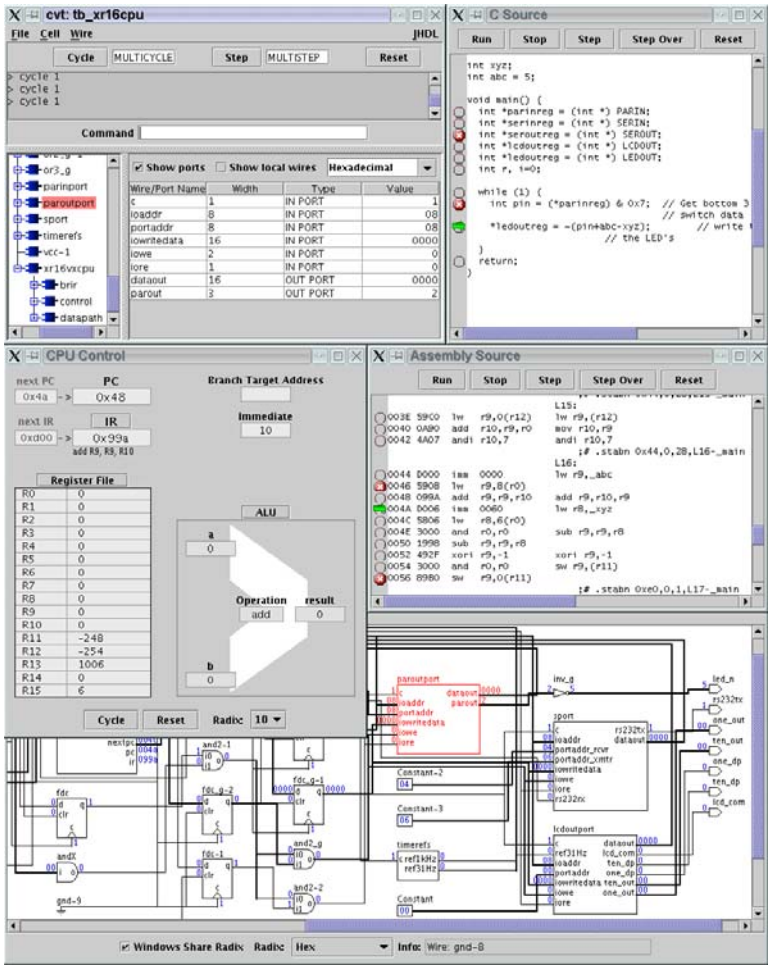


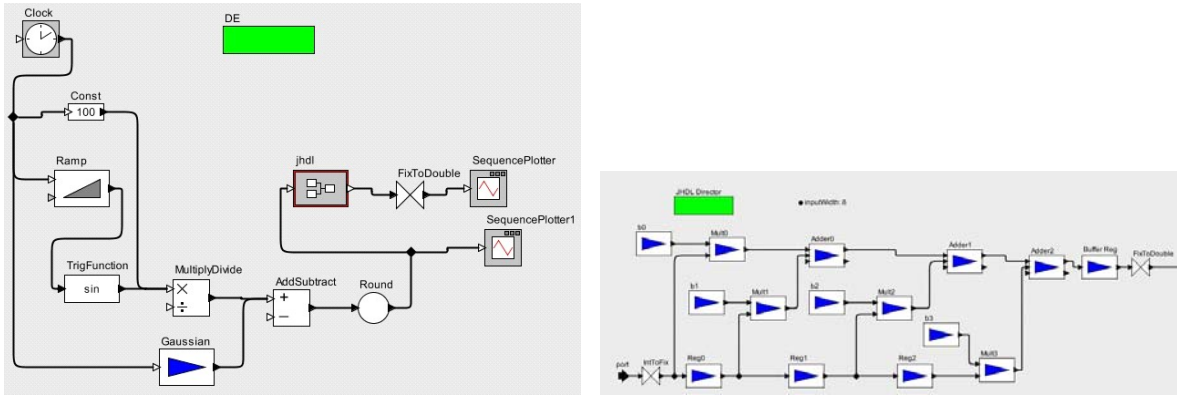
Figure 29: Prototype Debug Environment. Note that the CPU control window shows the currently executing instruction, while the source code windows reflect the instruction currently being decoded.

7.3 Integration of DSM to Ptolemy Modeling Environment

Another example of a high-level tool integrated with the DSM is the Ptolemy II heterogeneous modeling environment. This integration of Ptolemy II with the DSM allows the user to create models within Ptolemy that can be targeted to an FPGA platform [11]. Once these models have been created within Ptolemy, they can be executed on the ACS platform and take advantage of all the debugging capability provided by the DSM.

Figure 30 demonstrates a simple model created in Ptolemy using the JHDL libraries. The testbench for this model is shown in Figure 30 (a) and includes a number of debugging primitives such as a Gaussian random number generator and a sequence plotter. The JHDL model is shown in Figure 30 (b) and describes a simple FIR filter. This system model is decribed graphically using the Vergil visual programming tool provided by Ptolemy. Once defined within Ptolemy, the model is translated into a corresponding JHDL circuit within the DSM.

During system design and development, the designer will use the debugging aids provided by Ptolemy to verify the operation of the model. Once the model is verified and translated onto the



(a) Top-level Ptolemy Testbench Model

(b) Ptolemy Model Using JHDL Primitives

Figure 30: Example Ptolemy Model

FPGA, the user is able to use the *same* debugging tools found in Ptolemy to verify the hardware implementation of the system. One such debugging aid is a sequence plotter shown in Figure 31. In this example, the user evaluates the functionality of the system by evaluating the output of data sequences. Rather than forcing the user to use a different method to debug the system in hardware, integration of the DSM with Ptolemy allows this verification to take place using the same familiar Ptolemy debug aids.

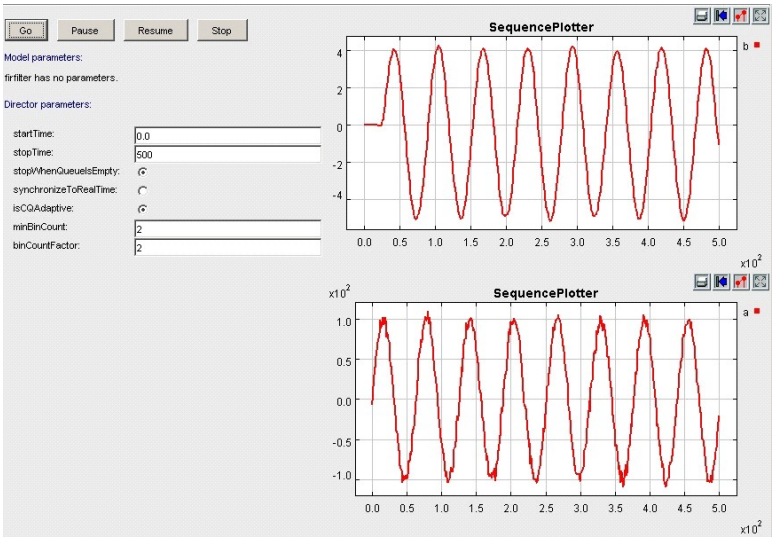


Figure 31: Output Ptolemy Sequence Ploter Using DSM

8 Use of the Unified Debug Environment on Real Applications

This section provides a description of our experiences using the Unified Debug Environment (UDE) on real applications. The first two were completed as a part of the DARPA-sponsored SLAAC project. As a part of that project, automated target recognition algorithms were being designed for FPGA-based execution. Versions of the UDE were used during much of that project to help in the debug of those designs. The third set of applications was completed as a part of this work specifically to demonstrate the flexible nature of the UDE for creating custom debugging environments.

8.1 Focus of Attention (FOA)

The Focus of Attention (FOA) project, used for automated target recognition, consists of two major components. First is the Adaptive Image Quantization (AIQ) unit, which produces intermediate quantized values for the pixels of a sonar image. Then the Binary Morphology circuitry makes use of spanning algorithms to produce the desired output data. For details refer to [6].

This design was targeted to the SLAAC1-V board for which the DSM provided a detailed board model. This proved to be valuable for debugging the FOA circuitry. The SLAAC1-V board model was integrated with a special GUI that simplified the process of configuring the FPGA's and loading the memories. More importantly however, was the fact that readback capabilities were also incorporated into the board model. The algorithms used in the FOA circuitry are computationally intensive and because each pixel of a given image is quantized and spanned, the entire process requires millions of clock cycles. Because of the large number of cycles required to produce the desired output data, debugging at hardware speeds became vital. This was possible due to the available readback capabilities provided by the board model and DSM. Through this medium, testing was able to be executed at hardware speeds and yet provide the visibility and control of the simulation environment. As a result, what could have taken months to debug was reduced to a few weeks.

8.2 Contamination Distribution Indexer (CDI)

The Contamination Distribution Indexer (CDI) is a template-based automatic target recognition (ATR) algorithm used by Sandia National Labs[20]. While this algorithm has been shown to be far superior than previous algorithms, it is more than an order of magnitute more computationally intensive than others. This algorithm was implemented on the SLAAC1-V board to demonstrate the advantages of ACS hardware.

Verification of the CDI implementation was a tedious and slow process due to the large number of test images and template data. Several million clock cycles were required to completely verify the operation of the circuit for a complete template/image pair. The debugging aids developed in this project were instrumental in the complete verification of CDI. Specifically, readback was used to provide a mixed hardware/software simulation of the complete system. Hardware execution was used quickly test the first set of test vectors. Once a known good state was reached, readback was used to view the system state within the DSM. The added visibility of the DSM helped identify a discrepency between the SLAAC1-V simulation model and the actual hardware execution. The use of the DSM avoided the cumbersome and time-consuming task of using a logic analyzer to trace board signals.

8.3 Custom Graphical Application Representation

Custom graphical *viewers* offer application-specific presentations of the circuit design. They also offer a portal through which the user may apply inputs without the need of being cognizant of the underlying design. Normally the DSM provides general-purpose circuit visualization including a standard set of viewers (e.g. schematic viewers, circuit hierarchy viewers, memory viewers, etc.) Each unique Adaptive Computing System (ACS) application may easily add to or remove from this default set of viewers as appropriate for the application. Each custom viewer merely needs to follow the DSM system's method for generating events.

Features that can be included within custom graphical viewers include: dynamic circuit model generation, netlisting, platform control, etc. This allows a generated design to be a small, simple, deployable CAD application. The end user needs only to execute the application in an appropriate runtime environment (i.e. ACS-specific drivers must be present, a Java runtime environment must be available, and FPGA-specific placement and routing tools may be required for design reconfiguration).

Custom graphical viewers provide a custom interface that best meets the needs of the application. The representation of a design need not be constrained to a pre-defined system like an ordinary CAD tool. Rather, the application is built on a foundation that offers the resources required to create a hardware configuration and corresponding model, and that allows the application to use its own custom interface.

An example using a custom graphical viewer described in [16]. The Edit Distance application described is a parameterizable JHDL circuit design. Program 8.1 shows a portion of the code for the EditDistance class that builds the circuit model. The parameters determine the circuit structure and function at circuit build time. The user provides the desired target string parameter when starting the application. Custom graphical viewers can more easily describe the impact of these build-time decisions.

Program 8.1 The EditDistance Class Which Creates the Circuit Model for the Edit Distance Application

```
public class EditDistance
      extends Logic
  public static CellInterface[]
    cell_interface =
      in("charIn",4),
      out("distanceOut",PARAM_WIDTH),
      param("outputWidth",INTEGER),
      param("targetString",STRING),
  ;
  public EditDistance(
        Node parent,
        Wire charIn,
        Wire distanceOut,
        String targetStr)
    // constructor code ...

// end class EditDistance
```

The graphical view for the Edit Distance application (Figure 32) provides a simple interface that

allows the user to input a source string, control the execution of the circuit, and view the progress of the string as it cycles through the processors of the systolic array. The inputs are in the form of text fields. The circuit visualization shows information about each processor in the circuit. On the top of the view of each processor is displayed the character that is embedded in that processor. Below that is a symbol that indicates whether the current source character in that processor is equal to the source string character or not. Below that is the character from the source string. Finally, on the bottom is the output value from the processor, which is passed down the array to the accumulator connected to the circuit output.

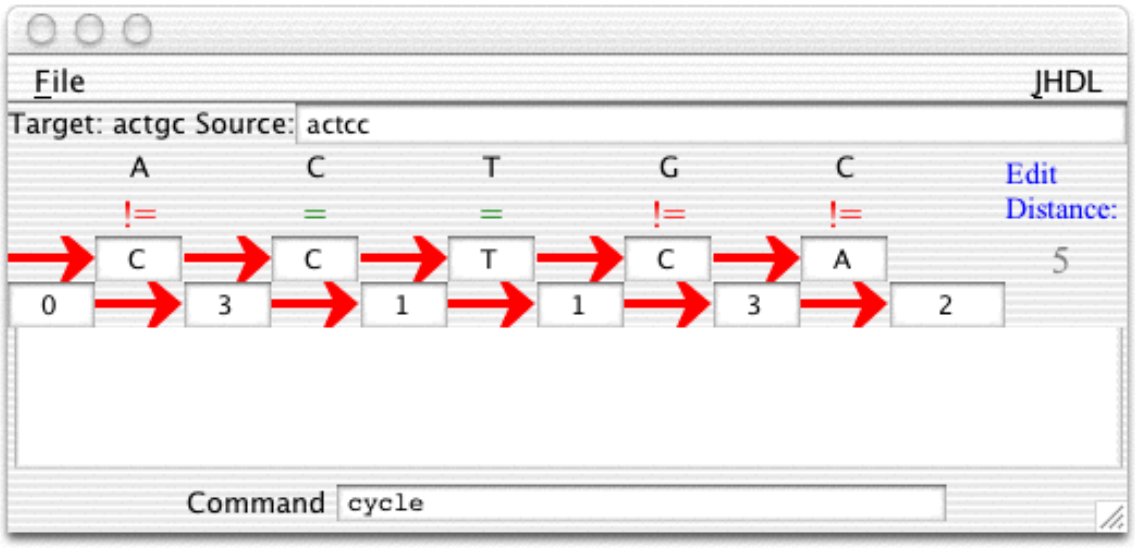


Figure 32: The Graphical User Interface for the Edit Distance Application

Each of the custom views in the edit distance application is a separate, parameterizable panel. The custom view of a single processor was used to help debug the hardware design. When the processor design was verified, both the processor and GUI subpanel were duplicated in parameterizable arrays in the hardware design and complete custom viewer respectively. Thus the development of the custom viewer not only built up the final application interface, it also aided in the hardware design and debug from the beginning of application development.

More infomation about custom GUI development can be found in paper[16] and the thesis by Slade [17].

9 On-line Documentation of the Unified Debug Environment

Documentation for the Unified Debuge Environemnt (UDE) can be found on the internet at the following location: http://www.jhdl.org/docs/docs/jhdlDocs.html. The starting document is pictured in Figure 33. Four main documents make up the documentation of the UDE: Overview, Getting Started, User's Manual, and the API Reference. Each of these are essential for understanding all of the UDE and JHDL.

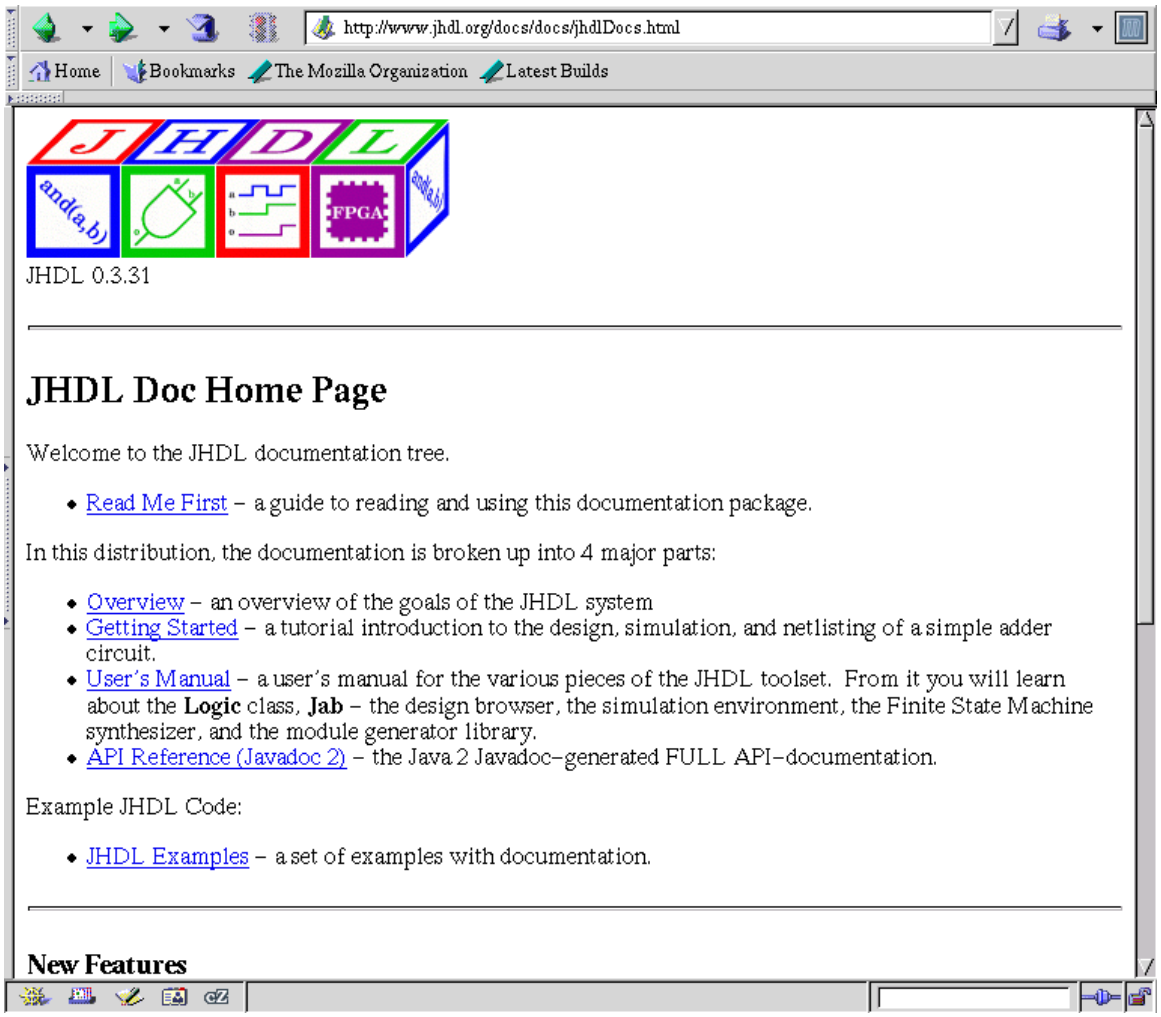


Figure 33: The JHDL Documentation Page

- AN OVERVIEW OF JHDL
  This document provides a broad overview of the history, the goals and the current state of JHDL. This document provides perspective into many of the design choices made in JHDL.

- THE JHDL GETTING STARTED GUIDE
  The Getting Started Guide explains how to create a design and simulate of a complete circuit. This guide provides an overview of the big picture and quickly shows how to use JHDL. Not all of the design elements are explained here, but enough to get a designer started.

- THE JHDL USERS MANUAL
  The JHDL Users Manual is meant to provide the in-depth discussion required to effectively use JHDL. The first three major sections are required reading before starting in on any real design using JHDL. First, *JHDL Cells and Wires* will discuss how JHDL elements (components and wires) are Java objects and may be instanced to create circuitry. The second section, *Introduction to Creating Logic Descriptions with JHDL*, and the four subsections on levels of design which follow it continue by teaching you how to create logic descriptions in JHDL using various libraries of building blocks. The third section descibes the use of clocks and how to master explicit clocking in JHDL. Further important sections discuss how to use JHDL for simulation, circuit browsing, and netlisting. Finally, advanced sections describe topics may not be immediately necessary, but which should be read as the need arises.

- THE API DOCUMENTATION
  JHDL contains a number of libraries and packages. The documentation on those libraries and packages must be consulted to learn how to use them. While the Users Manual sections talk about them, there simply is not enough information provided there to learn all of their uses. Collectively these sections of documentation are called the API Documentation. This documentation is generated using the Javadoc utility.

As the API Documentation is extensive and includes thousands of classes, its use will be described here. A screen-shot of the API documentation is given in Figure 34. To access the API Documentation for the Logic package do the following:

1. From the main JHDL Documentation Tree page you will see a link to the API Reference. Click it.

2. A frames-based Web page will come up. In the upper left corner you will see a list of Packages. Scroll down through the list. The interesting ones to you as a designer will be *byucc.jhdl.Logic, byucc.jhdl.Logic.Modules, byucc.jhdl.Xilinx.Virtex*.
   Click on *byucc.jhdl.Logic*.

3. In the lower left corner of your browser window, a list of classes from the selected package will appear. In this case, the interesting classes will be *Logic* and *LogicGates*. Click on *LogicGates*.

4. In the right half of your browser window the documentation for class *LogicGates* will appear. The interesting and useful part of this window is the Method Summary. As you can see, there are many, many methods in the class. These are the subroutines you can call to build circuitry. After scrolling through that, you will come to Method Detail. For each method in the class, this provides a detailed synopsis of the method. If you find a method you are interested in Method Summary, clicking on the method name will take you to its description in Method Detail.

5. Now go back to the lower left pane and click on the Logic class. In the right half of the screen you can scroll down through the summary of the methods in this class. After that you will come to a section labelled ”Methods inherited from class *byucc.jhdl.Logic.LogicGates*”. In this section are listed all the methods from the *LogicGates* class. Together, these routines and those defined in *Logic* itself make up the Logic API as described in the Users Manual.

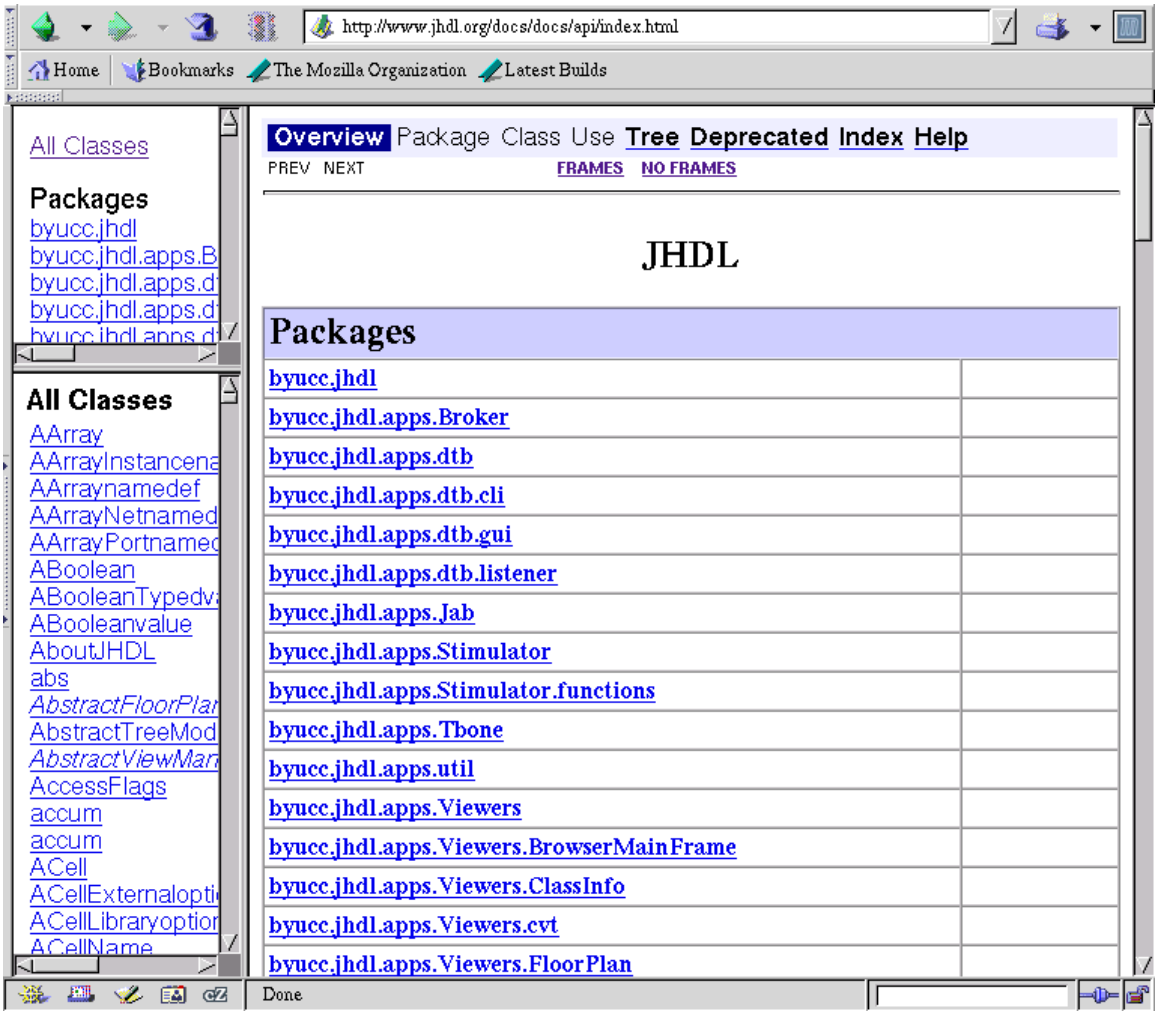


Figure 34: The JHDL API Documentation Page

10 Context Switching Devices Results Using the Sanders CSRC

The Sanders CSRC device is an integrated circuit implementing FPGA-like functionality. The main difference between it and conventional FPGA's is that it was designed to support the rapid switching between configurations (contexts). Using CSRC, a design can be done which consists of multiple contexts. The advantage is that a larger computation can be fit onto the device, provided it can be partitioned into mutually-exclusive contexts.

The debug challenges associated with CSRC were twofold: (1) creating a simulation environment which reflected the device's operation as it switched contexts and (2) creating a readback-like capability so that the executing hardware state could be extracted for back-annotation into the UDE. Work performed on these tasks is described in this section.

10.1 Investigate Context-Switched Approaches for Debug

One of the initial debug approaches considered for the CSRC context-switched device was the placement of debug circuitry in a dedicated context. This approach was ruled out when it was determined that when the context switches are data driven, the context running on the device is not externally available. It would not be possible for the host computer to switch out of the current context into the debug context and back without knowing which context to return to. The debug scheme chosen could not require any context switches, thus defeating the primary intent of the project.

In place of the dedicated context debug scheme, design-level scan was chosen. Design-level scan consists of adding multiplexors and gates to the memory elements of a design–such as flip-flops and embedded RAMs–so that the state elements' values can be serially shifted out of the FPGA. The main downside is that this added user circuitry might impose a high overhead to implement, even though it can be removed when debugging is complete. There are several benefits to using design-level scan. First, it can be added to any user design on the CSRC without additional hardware modifications. Secondly, the scan bitstream is generally small and easy to manipulate compared to a readback bitstream. Third, the circuit runs at full speed until the user is ready to take a snapshot of the circuit state, at which point the clock is still running in order to scan out the circuit state, although no useful work is being done by the circuit during those clock cycles. Fourth, scan allows the state of the circuit to be modified, providing the user with the ability to bring the circuit into a known state. Most of the code to implement scan was already a part of JHDL and only slight modifications were needed to integrate it with the CSRC library primitives.

During development, we were unable to actually run any scan-instrumented designs in hardware. This was due to the fundamental need to produce bitstreams from JHDL designs. Early problems with the back-end place and route tools prevented us from generating even the simplest bitstream through our toolpath.

10.2 Develop DSM support for CSRC

Support for simulation of and switching between multiple contexts was developed with the Reconfigurable Computing Module (RCM) board model and browser. It includes the capability to load an arbitrary number of designs into each CSRC simulation model and to visually designate the active context as shown in Figure 35.

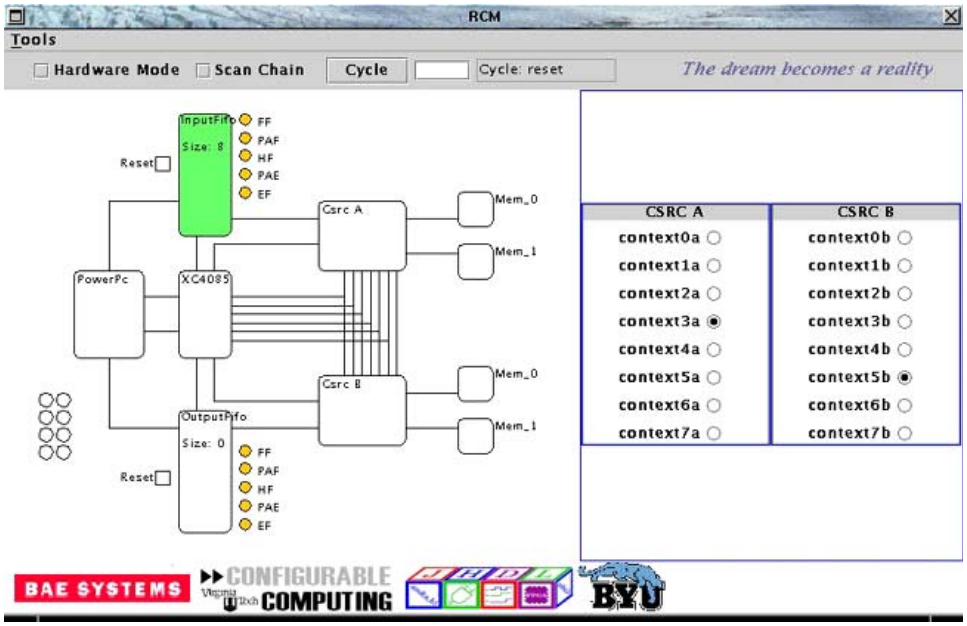


Figure 35: The Board Model With Numerous Contexts Ready for Simulation

The CSRC simulation model disables the clock for the inactive contexts, and enables it for the active context. A different context may be selected at any time during simulation. When the board model is in hardware mode, the same process is used to load bitstreams into the RCM board cache, and then to activate a particular context.

In addition, support for simulating data sharing between contexts was integrated into the board model. Data sharing allows a circuit designer to designate memory elements in different contexts (such as flip-flops) as ”shared.” When a context with shared memory elements is activated, it updates the internal state of the shared elements from a central data structure. Upon being deactivated, it writes the values of all shared elements out to the central data structure. In this way the shared data can be modified by one context and then read by a different context.

10.3 Test the CSRC Debug Support On CSRC-mapped Applications

Several simple applications such as adders, subtractors, and incrementers were developed in order to test the CSRC debug support. However, due to our inability to successfully place and route any designs, those applications could not be run in hardware and the debug support remains untested.

10.4 CSRC Debug Demonstration

This section will attempt to demonstrate what happens to a design when is instrumented with a scan chain. If you have loaded your design with a wrapper class, it will show up as shown in Figure 36 when you view it from the board model.

In the Figure, ”testcircuit” is the name of your actual design. It should be instanced like this in your wrapper class:

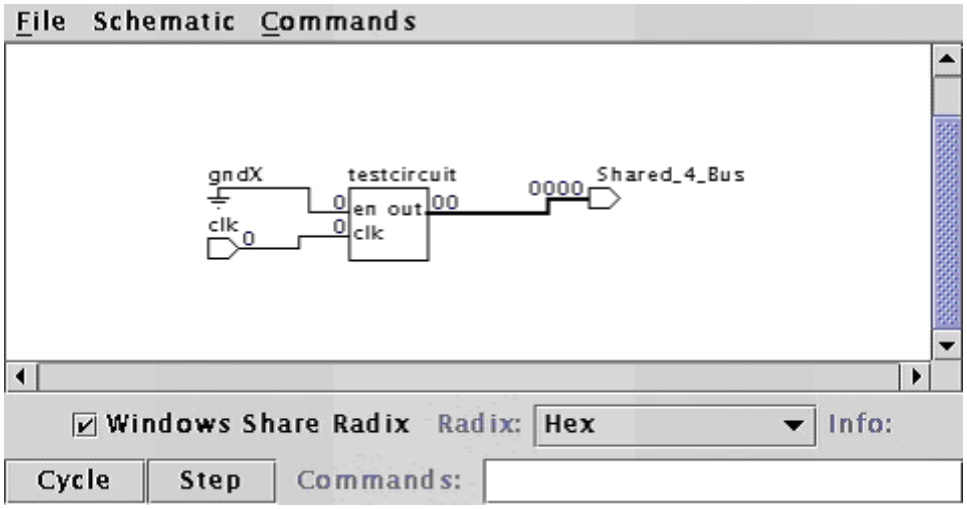


Figure 36: A Sample Circuit Loaded Into the Board Model

Cell my_circuit = new byucc.jhdl.platforms.rcm.testcircuit(this, gnd(), out, clock);

When you double-click on the ”testcircuit” component, you will push into the actual design. Figure 37 is what you should see, a schematic of the user design before it has been instrumented with scan chain.

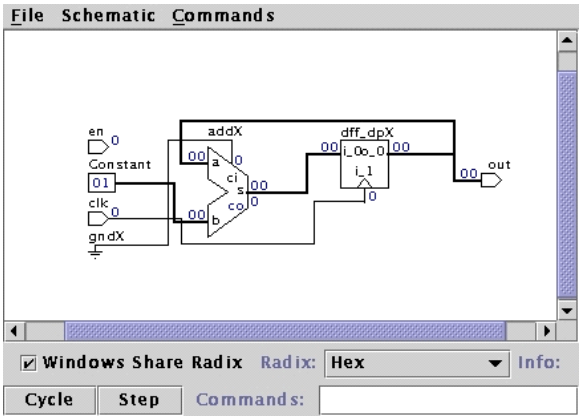


Figure 37: The View of the Sample Circuit After We Have Pushed Into ”testcircuit”

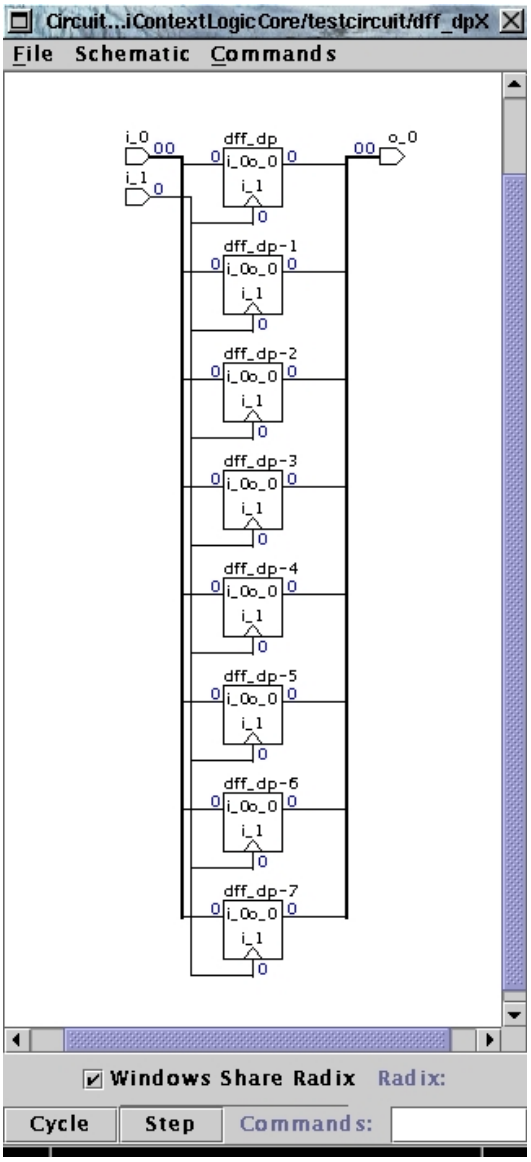


Figure 38: The Inside View of the Registers

We have a simple 8-bit adder whose output is registered. The inside of the register is shown in Figure 38 where ***i_o*** is the input, and ***i_1*** is the clock.

Now we instrument the design by using a wrapper class that implements scan chain. When we view it in the board model, it comes up as shown in Figure 39. Notice the addition of several components. First, we have the ControlScan module. This is used to control the input and output of the scan stream. Next, we have the module labeled SR_PISO. This is a parallel to serial converter, used to convert the parallel scan stream. Third, is the SR_SIPO. This is a serial to parallel converter for the scan stream. Notice also that we have a few additional ports that have been added to the testcircuit module. We will see what they do shortly, as we push into the test circuit as shown Figure 40.

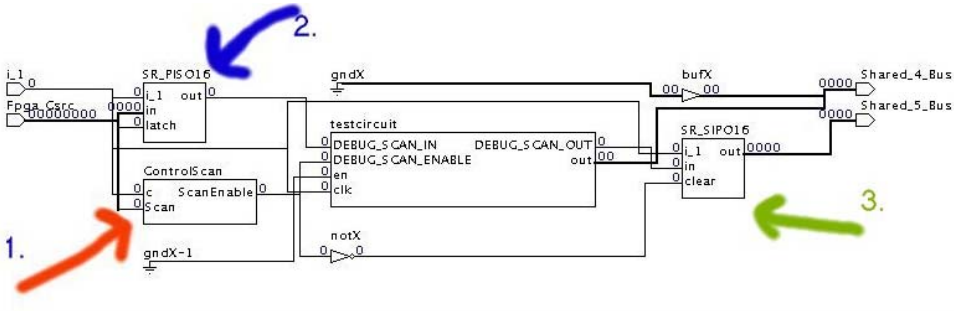


Figure 39: The High Level View of the Scan-Instrumented Circuit

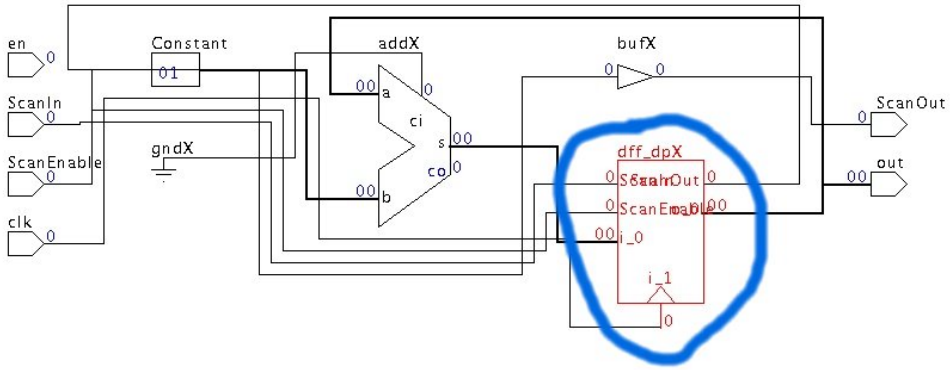


Figure 40: The ”testcircuit” Module After It Has Been Instrumented with Scan Chain

There are only a few differences between this and our original design. We have the ScanIn, ScanEnable, and ScanOut ports. These connect to the dff_dpX component. You may notice that the dff_dpX component has these additional ports as well, and has been substantially modified. The primary work of the scan chain takes place inside this new flip-flop, however, now it is much more than just flip-flops.

Finally, the inside of the dff_dpX is shown Figure 41. First of all we notice the addition of a 2:1 multiplexor for each of the individual flip-flops. Purple arrows indicate the location of three of these muxes. These muxes are controlled by the ScanEnable signal. When scan is disabled, the select line is low and the first input, i_0, is selected. When scan is enabled, the mux selects the other input and the flip-flops become a shift register, shifting the scan bitstream out and eventually back into the circuit to continue normal operation. Once the bitstream is shifted out, it is transferred using the RCMStreamData command back to the host. Host code then takes the bitstream and correlates each bit’s value with its flip-flop in the JHDL schematic. The actual hardware values are then displayed in the JHDL schematic viewer, and the original bitstream is shifted back into the chip to restore the original state of the circuit.

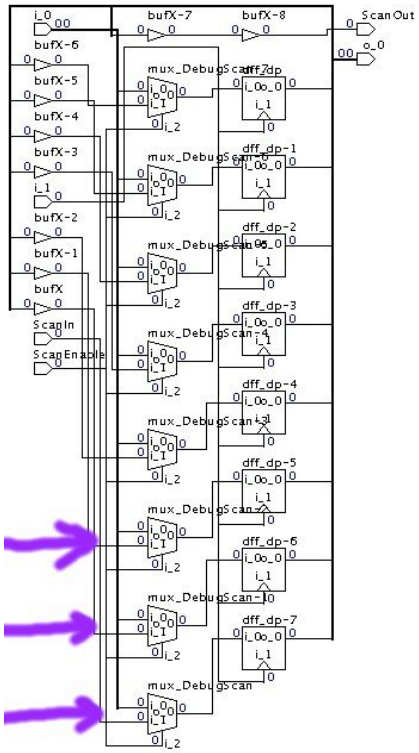


Figure 41: Muxes Are Added to each Flip-Flop

11 Conclusions and Future Work

A Unified Debug Environment for Adaptive Computing Systems was created in this work. This provides a number of advantages over current debug systems. The most important is that it provides a unified simulation/hardware execution environment for debugging and verifying ACS designs. This is in contrast to existing systems which separate simulation and hardware debug, requiring different tools for each.

The core of the Unified Debug Environment is the DSM. This provides support for a number of additional features in debugging ACS designs. The first of these is checkpointing. This allows the state of an executing ACS application to be captured, saved, and then restored at a later time. Using this capability, multi-tasked execution of multiple designs on a single ACS platform was demonstrated. A second feature of the DSM is its support for remote execution. Using remote procedure calls, the DSM allows users at remote sites to download and debug designs on an ACS platform. A third feature of the DSM is its support for automatic synthesis of debug circuitry. Using this, a designer can interactively modify his design for debug (configure embedded logic analyzers, change register values, etc). This provides a capability similar to that provided for software debug and greatly enhances ACS debug efficiency. Finally, the DSM provides the ability to interface with external high-level tools. This is done using its EDIF import and export capability as well as its open-API structure.

11.1 Future Work

The goal of the project PI's has been to provide, as far as is possible, debug capabilities for ACS similar to those already available for software debug. The unified simulation/hardware execution capabilities of the UDE and the range of services provided by the DSM form the foundation for doing this.

However, much remains to be done and this work is only a beginning. Debug for ACS has traditionally been viewed by the research community as decoupled from the design process. It has been an afterthought. The majority of ACS research in the community has focused on design entry, specifically attempting to make it possible for non-hardware designers to design for ACS.

We view the deployment of ACS applications as consisting of three phases or steps: design, debug, and deployment as described in [17]. Design of ACS applications for non-hardware designers remains a problematic area. In this work a lesson learned is that debug cannot be decoupled from design. In software compilation, a significant amount of information is recorded during the compilation process solely for the purpose of simplifying the debug activity that will surely follow. As shown in [3] simply obtaining the equivalent information for an FPGA design is a daunting task. The creation of standard file formats for recording such information would go a long way toward solving many of the problems associated with provided a tight coupling between the design and debug of ACS applications.

The goal of this work was to produce general ACS debug techniques and methodologies which would work across a range of design tool flows. However, our experience in this work was that most of the techniques we demonstrated could not have been simply "bolted onto" an existing design and simulation environment. Indeed, the majority of the interesting features of the UDE (checkpointing, multitasking, debug circuitry synthesis, and unified simulation/hardware debug) were only possible

because we controlled the JHDL source code base and could modify it as needed to provide the DSM functionality desired.

Thus, while this work has demonstrated a wide range of ACS debugging features and techniques, these are not compatible with current VHDL or Verilog design environments. To fully take advantage of what was demonstrated in this work, design tool environments need to build in DSM-like support from the outset.

Runtime environments for ACS which provide features similar to those provided for conventional computing systems are still relatively new ideas. Our view is that the deployment phase of ACS applications requires (or can certainly benefit from) most of the UDE and DSM functionality described in this report. Slade, in [17], argues this and goes on to describe how the UDE and DSM can be used to create custom computing environments for deploying ACS applications. His thesis is that the UDE/DSM *becomes* an integral part of the final application and parts of the UDE are distributed along with the final application. The demonstrations performed as a part of this work and documented in this report amply demonstrate this point.

Since the commencement of this research the number of high-level design methodologies for ACS has grown beyond simply using VHDL or Verilog to do hardware design. Examples include C-based compilation approaches, Matlab compilation approaches, and graphical approaches. Some of these are even now available as commercial products from companies such as Xilinx, Celoxica, and AccelChip. As such tools make ACS accessible to a wider variety of users (specifically non-hardware engineers) the availability of advanced debug tools will become more and more important.

12 References

[1] J. M. Arnold. The Splash 2 software environment. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 88–93, Napa, CA, April 1993.

[2] P. Graham, B. Nelson, and B. Hutchings. Instrumenting bitstreams for debugging FPGA circuits. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'2001)*, April 2001.

[3] Paul Graham, Brad Hutchings, and Brent Nelson. Improving the FPGA design process through determining and applying logical-to-physical design mappings. In Brad L. Hutchings, editor, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 305–306, Napa, April 2000. IEEE Computer Society, IEEE Computer Society Press.

[4] Paul S. Graham. *Logical Hardware Debuggers for FPGA-Based Systems*. PhD thesis, Brigham Young University, Dept. of Electrical and Computer Engineering, 2001.

[5] K. Scott Hemmert and Brad Hutchings. Issues in debugging highly parallel FPGA-based applications derived from source code. In *Proceedings Asia and South Pacific Design Automation Conference 2003*, pages 483–488, Kitakyushu, Japan, 2003.

[6] K. Scott Hemmert, Brad L. Hutchings, and Anshul Malvi. An application-specific compiler for high-speed binary image morphology. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, IEEE Computer Society Press, April 2001.

[7] K. Scott Hemmert, Justin L. Tripp, and Brad L. Hutchings. Source level debugger for the sea cucumber synthesizing compiler. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 228–237, Napa, CA, 2003. IEEE.

[8] Karl S. Hemmert. *Issues in Source Level Debugging of Circuits Synthesized for Configurable Logic*. PhD thesis, Brigham Young University, 2003.

[9] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting. A CAD suite for high-performance FPGA design. In K. L. Pocek and J. M. Arnold, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 12–24, Napa, CA, April 1999. IEEE Computer Society, IEEE.

[10] Brad L. Hutchings and Brent E. Nelson. Using general-purpose programming languages for FPGA design. In *37rd Design Automation Conference (DAC)*, pages 561–566, Los Angeles, CA, June 2000.

[11] Matthew Koecher. Hardware synthesis of synchronous data flow models. Master's thesis, Brigham Young University, Department of Electrical and Computer Engineering, December 2003.

[12] Wes Landaker, Michael Wirthlin, and Brad Hutchings. Multitasking hardware on the SLAAC1-V reconfigurable computing system. In *Field-Programmable Logic and Applications. Proceedings of the 12th International Workshop, FPL 2002*, Lecture Notes in Computer Science, pages 806–815. Springer-Verlag, 2002.

[13] Wesley J. Landaker. Using hardware context-switching to enable a multitasking reconfigurable computer system. Master's thesis, Brigham Young University, Department of Electrical and Computer Engineering, August 2002.

[14] E. Roesler and B. Nelson. Debug Methods for Hybrid CPU/FPGA Systems. In *Proceedings of The First IEEE International Conference on Field-Programmable Technology (FPT)*, pages 243–251, December 2002.

[15] Eric E. Roesler. A simulation and hardware execution co-verification environment for soft-core cpus. Master's thesis, Brigham Young University, Department of Electrical and Computer Engineering, August 2003.

[16] A. Slade and B. Nelson. Reconfigurable Computing Application Frameworks . In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '03)*. IEEE Computer Society, IEEE Computer Society Press, April 2003.

[17] Anthony Lynn Slade. Designing, debugging, and deploying configurable computing machine-based applications using reconfigurable computing application frameworks. Master's thesis, Brigham Young University, Department of Electrical and Computer Engineering, April 2003.

[18] Justin L. Tripp, Preston A. Jackson, and Brad L. Hutchings. Sea Cucumber: A synthesizing compiler for FPGAs. In *Field-Programmable Logic and Applications*, pages 875–885. Springer, September 2002.

[19] Timothy Wheeler, Paul Graham, Brent Nelson, and Brad Hutchings. Using design-level scan to improve FPGA design observability and controllability for functional verification. In *Field-Programmable Logic and Applications. Proceedings of the 11th International Workshop, FPL 2001*, Lecture Notes in Computer Science, pages 483–492. Springer-Verlag, August 2002.

[20] Michael J. Wirthlin, Steve Morrison, Paul Graham, and Brian Bray. Improving performance and efficiency of an adaptive amplification operation using configurable hardware. In K. L. Pocek and J. M. Arnold, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 267–275, Napa, CA, April 2000. IEEE Computer Society, IEEE.

A The DSM API's

A.1 Debug Service Manager

A.1.1 The Simulator

- ***void initialize()*** - initializes the simulator.
- ***void haltSimulator()*** - stops simulation.
- ***int getSimulationErrorCount()*** - This returns the number of floating or multiple driver errors on the last simulation run.

A.1.2 The Hardware System (HWSystem)

- ***void addObservable(Observable obs)*** - Adds an Observable to the circuit.
- ***void addSimulatorCallback(SimulatorCallback sc)*** - Adds a SimulatorCallback to the circuit.
- ***protected boolean check()*** - Returns true if this node was valid.
- ***boolean collectTransitionCountEnabled()*** - This method returns true if transition counts have been enabled, false if they have not been set.
- ***boolean collectTransitionCountEnabled(boolean flag)*** - This method enables (true) or disables (false) the transition counts in the current HWSystem.
- ***void cycle(int no_clocks)*** - This method is invoked to advance the simulator by no_clocks.
- ***boolean cycleHardware(int no_steps)*** - Runs a hardware platform, provided the system is in hardware mode and the root node implements HardwareInterface.
- ***Wire declareExternalClock(Wire clock_wire)*** - This declares a wire to be an external clock.
- ***void disableAllHWUpdate()*** - As a matter of convenience, this will disable all HWUpdating of all Clockable cells returning to pure simulation.
- ***void enableProgrammaticTestBench()*** - Called if we want to use programmatic testbenches.
- ***Nameable findNamed(String name)*** - Used to search the circuit graph for a node or wire by name.
- ***boolean findPropagated(Wire w)*** - Finds the propagated wire in the global schedule.
- ***void forceSimulatorCallbackRefresh()*** - This method can be used to force a call to the sim_update of all SimulatorCallbacks and Observables.
- ***void forceSimulatorInitialize()*** - Forces a simulator (re)initialization.
- ***boolean forceUniquify()*** - Returns the status of system-wide uniquification.
- ***boolean forceUniquify(boolean flag)*** - Sets the status of system-wide uniquification.

- ***Checkpointable[] getCheckpointableCells()*** - Method to get the list of all CheckpointableCells.
- ***int getClockValue(Wire clock)*** - Convenient for getting the current clock value.
- ***int getClockValue(Wire clock, int future_step)*** - This function exists to allow the user access to the values of the clock schedule.
- ***int getCurrentStepCount()*** - Gets the current step count.
- ***String getDefaultClockSchedule()*** - In some cases, users may want to see what the current schedule is for the default clock.
- ***ExternallyUpdateable[] getExternallyUpdateableCells()*** - Method to get the list of ExternallyUpdateableCells.
- ***boolean getHardwareMode()*** - Whether the simulator is in hardware mode.
- ***LargeExternallyUpdateable[] getLargeExternallyUpdateableCells()*** - Method to get the list of all LargeExternallyUpdateableCells.
- ***int getLastClockValue(Wire clock)*** - Convenient for getting the prior clock value.
- ***int getNextClockValue(Wire clock)*** - Convenient for getting the next clock value.
- ***int getStepsPerClock()*** - Returns the number of steps per clock.
- ***Cell getTestBench()*** - A HWSystem should have only one child and it should be a test bench, based on constructors.
- ***int getTotalClockCount()*** - Simple accessor.
- ***CellList getUnscheduledSourceCells()*** - Returns the unscheduled cells.
- ***boolean getUsingImplicitClock()*** - Gets the usingImplicitClock flag.
- ***boolean hardwareModeSane()*** - Whether the simulator is in hardware mode and a valid hardware interface exists.
- ***void initialize()*** - This will force an initialize of all of the internal data structures of the HWSystem at the appropriate time (it may not happen immediately).
- ***boolean isRoot()*** - Returns true, as the HWSystem is the root.
- ***boolean isSimulatorInitialized()*** - Whether the simulator is initialized.
- ***void libraryDefaultClock()*** - This should only be called by libraries.
- ***boolean multiClockModeEnabled()*** - Whether the circuit is in multi-clock mode.
- ***boolean netlistCheck()*** - Checks that the circuit is valid for netlisting.
- ***void optimizeForSimulation()*** - Optimizes the circuit for simulation.
- ***void printSimulatorInternals(OutputStream output_stream)*** - Prints the status of the simulator.

- ***long printTransitionCounts(OutputStream os)*** - This will print out all of the transition counts for every wire in the current testbench and it's corresponding children.
- ***long printTransitionCounts(OutputStream os, Cell c)*** - This will print out all of the transition counts for every wire in the target Cell and it's children.
- ***void readSystemState(InputStream is)*** - This is to cause the ExternalUpdate Manager to load the state of system from the State object corresponding to the given cycle count.
- ***void readSystemState(int cycle, int step)*** - This is to cause the ExternalUpdate Manager to load the state of system from the State object corresponding to the given cycle count.
- ***void readSystemState(String name, int cycle, int step)*** - This is to cause the ExternalUpdate Manager to load the state of system from the State object corresponding to the given cycle count.
- ***void removeSimulatorCallback(SimulatorCallback sc)*** - Adds a SimulatorCallback to the circuit.
- ***void reset()*** - This will reset the simulation.
- ***void setDefaultClockSchedule(String schedule)*** - Allows users to modify the current default clock schedule.
- ***String setHardwareMode(boolean value)*** - This tells that simulator that you will or will not be using the hardware versus the simulator.
- ***void setUsingImplicitClock()*** - Sets the usingImplicitClock flag.
- ***void skip(int no_steps)*** - This method is invoked to advance the simulator by no_ clocks.
- ***void step(int no_steps)*** - This method is invoked to advance the simulator by no_ steps.
- ***void updateHardware()*** - Performs a hardware readback without stepping the hardware clock.
- ***void useHWUpdate(Clockable cell, boolean flag)*** - This tells the simulator that you will use HWUpdate mode to update the state of the Clockable cell you pass as the argument.
- ***void writeSystemState()*** - This is to cause the ExternalUpdate Manager to dump the current state of the system to a StateObject...
- ***void writeSystemState(OutputStream os)*** - This is to cause the ExternalUpdate Manager to dump the current state of the system to a StateObject.
- ***void writeSystemState(String name)*** - This is to cause the ExternalUpdate Manager to dump the current state of the system to a StateObject.
- ***void writeSystemStateToHardware()*** - Writes the simulator system state to hardware, if the current loaded model allows it.

A.1.3 Wire Value Interface

- ***int get(Cell sink)*** - Values are read from wires using this method, where the sink is the node being driven by the wire.
- ***void put(Cell source, int val)*** - Values are placed on a wire using this method, where the source is the node driving the wire.

A.1.4 Circuit Structure

- ***void addPort(CellInterface portInterface)*** - Adds a port to the cell, after it has been created.
- ***void addPorts(CellInterface[] portInterfaces)*** - Adds an array of ports to the cell, after it has been created.
- ***void bind(String name, boolean value)*** - Binds a boolean parameter.
- ***void bind(String name, int value)*** - Binds an integer parameter.
- ***void bind(String name, long value)*** - Binds a long parameter.
- ***void bind(String name, Object value)*** - Binds a declared generic variable with a value.
- ***Wire connect(String portname, Wire w)*** - Attaches the wire to an input or output port by name, using the information in portios to determine the direction of the port.
- ***String getCellName()*** - Access the cell name associated with a derived class.
- ***NodeList getChildren()*** - Returns the children of this node.
- ***Wire getDefaultClock()*** - This returns the default clock of this level of hierarchy.
- ***CellList getDescendents()*** - Returns a CellList containing all descendents of this cell.
- ***Node getParent()*** - Returns the parent of this Node.
- ***Cell getParentCell()*** - Returns the parent of this Node, cast as a Cell if possible.
- ***WireList getSinkWires()*** - Returns a list containing all of the sink wires for this Cell.
- ***WireList getSourceWires()*** - Returns a list containing all of the source wires for this Cell.
- ***HWSystem getSystem()*** - Returns the system.
- ***WireList getWires()*** - Returns the wires created by this node.
- ***static CellInterface in(String name, int width)*** - Creates the declaration for an in-port, to be used in the CellInterface[].
- ***static CellInterface in(String name, String width)*** - Creates the declaration for an in-port, to be used in the CellInterface[]
- ***static CellInterface inout(java.lang.String name, int width)*** - Creates the declaration for an inout-port, to be used in the CellInterface[].

- ***static CellInterface inout(String name, String width)*** - Creates the declaration for an inout-port, to be used in the CellInterface[].
- ***static CellInterface out(String name, int width)*** - Creates the declaration for an out-port, to be used in the CellInterface[].
- ***static CellInterface out(String name, java.lang.String width)*** - Creates the declaration for an out-port, to be used in the CellInterface[].
- ***static CellInterface param(String name, Class type)*** - Creates the declaration of a Parameter, to be used in the CellInterface[].
- ***void removePort(String portname)*** - Removes a port from the cell, after it has been created.

A.2 Platform Control Interface

A.2.1 Hardware Interface

- ***StateObject getHardwareState(ExternallyUpdateable[] eCells, LargeExternallyUpdateable[] leCells, Checkpointable[] cCells)*** - Returns the hardware state.
- ***void stepHardwareClock(int cycles)*** - Steps the hardware clock.

A.2.2 Hardware Controller Interface

- ***int close()*** - Closes the board.
- ***void finalize()*** - Prepares to close the board?
- ***int freeRunClock(int clock)*** - Free runs the specified clock.
- ***int getMemory(int set, int memory, int address, int length, byte[][] data)*** - Gets the contents of the specified memory.
- ***int getMemoryWidth(int set, int memory)*** - Returns the width of the specified memory element.
- ***int getRegister(int set, int register, byte[] data)*** - Returns the width of the specified memory element.
- ***int getRegisterWidth(int set, int register)*** - Returns the width of the specified memory element.
- ***int open()*** - Opens the board.
- ***int program(int fpga, byte[] data)*** - Programs the specified FPGA.
- ***int readback(int fpga, byte[] data)*** - Readbacks the state of the specified FPGA.
- ***int setClockFrequency(int clock, float freq)*** - Sets the frequency of the specified clock.
- ***int setMemory(int set, int memory, int address, int length, byte[][] data)*** - Sets the contents of the specified memory.

- ***int setRegister(int set, int register, byte[] data)*** - Returns the width of the specified memory element.
- ***int stepClock(int clock, int steps)*** - Steps the specified hardware clock the given number of cycles.
- ***int stopClock(int clock)*** - Stops the specified free-running clock.
- ***int writeback(int fpga, byte[] data)*** - Writeback the state of a circuit.

A.2.3 Readback Manager

- ***void addReadBack(String fileName, String hierName, int peNum)*** - Initializes the readback data structures for a specific PE.
- ***void disableReadBack(int peNum)*** - Disables readback for a specific PE on subsequent calls to performReadBack().
- ***void enableReadBack(int peNum)*** - Enables readback for a specific PE on subsequent calls to performReadBack().
- ***int[] getEUValues(ExternallyUpdateable[] eCells)*** - Provides an array of readback values corresponding to the given array of ExternallyUpdatable Cells.
- ***int[][] getLEUValues(LargeExternallyUpdateable[] leCells)*** - Provides an array of readback values corresponding to the given array of LargeExternallyUpdatable Cells.
- ***void performReadBack()*** - Causes the state of all enabled PEs to be sampled.
- ***void removeReadBack(int peNum)*** - Reinitializes the readback data structures to a ”clean” state.
- ***void setReadBackEnable(int peNum, boolean enabled)*** - Sets the readback enable for the given PE to the provided value.